



全美经典  
学习指导系列

# C++ 编程 习题与解答

PROGRAMMING WITH C++

最佳的复习资料，实用的辅助教材

与国外高校计算机水平保持同步

为考研和出国深造奠定坚实基础

John R. Hubbard 著  
陈建江 王栋 何路 等译

Mc  
Graw  
Hill

机械工业出版社  
China Machine Press

中国出版社  
China National Press

全球销售超过  
3000 万册！

全美经典学习指导系列是一套快捷有效的学习指南,该套丛书针对各专业的技术重点提供了数百个实例、习题及答案。通过这些实战练习,不但可以洞悉各门技术精髓,而且能够使考试成绩大幅攀升,更会助你与国外大学生的计算机水平看齐,为将来考研或出国深造奠定坚实基础。

全美经典学习指导系列深得高校学生的喜爱。由于有了这套丛书,在历年的专业考试中,成千上万的学生获得了优异成绩。想成为一名优等生吗?——请选择全美经典学习指导系列!如果时间不裕却想成绩骄人,这本书可以助你:

- 通过具体范例解决疑难问题
- 考前快速强化
- 迅速找到答案
- 快捷而高效地学习
- 迅速掌握技术重点,无需翻阅冗长的教科书

全美经典学习指导系列以方便快捷的形式提供了考生需要了解的信息,同时不致使你淹没在不必要的细节当中。另外,还可以通过大量的编程练习来测试所学的技巧。该丛书可以与任何教材配合使用,使学生们能够根据各自的进度来学习,从而获得事半功倍的效果!全美经典学习指导系列的内容系统而完备,是毕业考试和专业考试的理想参考书。

本书包括:

- 涵盖了《计算机科学 I·II》——83%的大学使用的计算机语言
- 对递归、逻辑、多态性、算法等进行了简要的解释
- 解答了470余个C++方面的实例,包括详细步骤的说明
- 大量的例题及习题解答会帮助你掌握C++基础

如果想获得优异成绩并且能够全面掌握C++基础,本书是不可或缺的最佳辅导老师

C++ 编程习题与解答

C++ 编程习题与解答 (英文版·第2版)

Java 编程习题与解答

Java 编程习题与解答 (英文版)

SQL 编程习题与解答

Visual Basic 编程习题与解答

操作系统习题与解答 (英文版)

关系数据库习题与解答

关系数据库习题与解答 (英文版)

计算机导论习题与解答

计算机导论习题与解答 (英文版)

计算机体系结构习题与解答 (英文版)

计算机图形学习题与解答

计算机图形学习题与解答 (英文版·第2版)

计算机网络习题与解答

软件工程系习题与解答

数据结构习题与解答

——Java 语言描述

数据结构习题与解答

——Java 语言描述 (英文版)

数据结构习题与解答

——C++ 语言描述 (英文版)

封面设计  
江丽萍

ISBN 7-111-10821-3



网上购书: [www.china-pub.com](http://www.china-pub.com)

北京市西城区百万庄南街1号 100037  
购书热线: (010)68995259, 8006100280  
(北京地区)  
总编信箱: [chiefeditor@hzbook.com](mailto:chiefeditor@hzbook.com)



中国出版

ISBN 7-111-10821-3/TP·2589


定价: 39.00元


全美经典  
学习指导系列

# C++ 编程 习题与解答

PROGRAMMING WITH C++

John R. Hubbard 著  
徐漫江 王栋 何路 等译

 机械工业出版社  
China Machine Press

 中信出版社  
GUANGDONG PUBLISHING HOUSE

本书由浅入深地介绍了C++语言的各个方面，并在所涉及的各个知识点给出了详细的例子，使读者能够更容易了解C++的内容。无论读者是从未接触过C++语言的新手，还是对C++语言有一定经验的开发人员，本书都能使你对这门编程语言有全面而系统的了解。

John R. Hubbard: Programming With C++, Second Edition.

Copyright © 2000 by The McGraw-Hill Companies, Inc.

Original language published by The McGraw-Hill Companies, Inc. All Rights reserved.  
No part of this publication may be reproduced or distributed in any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

Simplified Chinese translation edition jointly published by McGraw-Hill Education (Asia) Co. and China Machine Press & CITIC Publishing House.

本书中文简体字翻译版由机械工业出版社、中信出版社和美国麦格劳-希尔教育(亚洲)出版公司合作出版。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有McGraw-Hill公司防伪标签，无标签者不得销售。

版权所有，侵权必究。

本书版权登记号：图字：01-2001-5204

### 图书在版编目(CIP)数据

C++编程习题与解答/(美)哈伯德(Hubbard, J. R.)著;徐漫江等译.-北京:机械工业出版社,2002.8

(全美经典学习指导系列)

书名原文:Prorgamming with C++, Second Edition

ISBN 7-111-10821-3

I. C… II. ①哈… ②J… ③徐… III. C语言-程序设计-解题 IV. TP312.44

中国版本图书馆CIP数据核字(2002)第062694号

机械工业出版社(北京市西城区百万庄大街22号 邮政编码100037)

责任编辑:华章

北京忠信诚印刷厂印刷·新华书店北京发行所发行

2002年8月第1版第1次印刷

787mm×1092mm 1/16·28印张

印数:0 001-5 000册

定价:39.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换



# 作 者 序

像所有的 Schaum 系列丛书中的书籍一样,本书主要是为方便读者自学而编写的,适合与一门正规的C++ 编程语言或计算机科学的课程一同使用。同时也很适于独立学习或作为一本参考书使用。

本书包括了 200 多个精彩例题和精心组织的有详细解答的习题。通过学习书中精心编写并做了详尽解释的例题,读者可以熟练掌握数据结构的基本原则,这也正是本书的编写意图。

C++ 是早在 20 世纪 80 年代由 Bjarne Stroustrup 开发的,它基于 C 和 Simula,是当今最流行的面向对象的编程语言之一。在 1998 年,美国国家标准化协会(ANSI)和国际标准化组织(ISO)为该语言制定了标准,这个新的 ANSI/ISO 标准包含了功能强大的标准模板库(STL),本书是完全遵照该标准而编写的。

虽然很多要学习 C++ 的人已经有了一些编程经验,作者还是假定本书的读者没有任何编程经验,而将 C++ 看做是读者所学习的第一门编程语言。这样,那些具有编程经验的读者可以跳过前面的几章。

C++ 是一门很难学的语言,原因至少有两个:第一,它继承了 C 语言的表达体制,很多含义很模糊;第二,作为一门面向对象的编程语言,普遍使用了类和模板的概念,这对以前没有这方面思想的读者而言是个极大的挑战。本书的编写意图是为那些刚刚起步的程序员们克服这些障碍而提供所需的帮助。

本书中的所有例题和问题,包括附加问题的源代码,都可以从以下网址下载:<http://projectEuclid.net/schaums>, <http://www.richmond.edu/~hubbard/schaums>, <http://hubbards.org/schaums>, <http://jhubbard.net/schaums>。另外,本书的修正和附录也可以在这些站点找到。

我想感谢所有的朋友、同事、学生和为本书提供了许多批评和建议的 McGraw-Hill 的人员,包括 John Aliano、Arthur Biderman、Francis Minhthang Bui、Al Dawson、Peter Dailey、Mohammed El-Beltagy、Gary Galvez、Libbie Geiger、Sergei Gorlatch、Chris Hames、John Troncale、Maureen Walker、Stefan Wentzig 和 Nat withers,在此对他们的建议和所做的调试工作表示衷心的感谢。

特别感谢我的妻子和工作伙伴 Anita H. Hubbard,感谢她为本书所提供的建议、鼓励和创造性意见,书中的许多源程序都是她编写的。

# 目 录

第1章 C++ 程序设计基础 .....	( 1 )
1.1 入门 .....	( 1 )
1.2 程序实例 .....	( 2 )
1.3 输出运算符 .....	( 4 )
1.4 字符与文字 .....	( 5 )
1.5 变量及其他声明 .....	( 5 )
1.6 程序标记 .....	( 6 )
1.7 初始化变量 .....	( 7 )
1.8 对象、变量和常量 .....	( 8 )
1.9 输入运算符 .....	( 9 )
复习题 .....	( 9 )
习题 .....	( 11 )
复习题答案 .....	( 12 )
习题答案 .....	( 13 )
第2章 基本类型 .....	( 18 )
2.1 数值数据类型 .....	( 18 )
2.2 布尔型 .....	( 19 )
2.3 枚举型 .....	( 19 )
2.4 字符型 .....	( 21 )
2.5 整型值 .....	( 22 )
2.6 数学运算符 .....	( 23 )
2.7 增量运算符和减量运算符 .....	( 24 )
2.8 组合赋值运算符 .....	( 24 )
2.9 浮点型 .....	( 25 )
2.10 类型转换 .....	( 27 )
2.11 数值溢出 .....	( 29 )
2.12 舍入错 .....	( 30 )
2.13 浮点数的电子格式 .....	( 33 )
2.14 作用域 .....	( 34 )
复习题 .....	( 35 )

习题 .....	(35)
复习题答案 .....	(36)
问题答案 .....	(36)
<b>第3章 选择</b> .....	(39)
3.1 If 语句 .....	(39)
3.2 if..else 语句 .....	(40)
3.3 关键字 .....	(40)
3.4 比较运算符 .....	(41)
3.5 语句块 .....	(43)
3.6 复合条件 .....	(44)
3.7 短路 .....	(45)
3.8 布尔表达式 .....	(46)
3.9 选择的嵌套 .....	(47)
3.10 else if 结构 .....	(50)
3.11 switch 语句 .....	(51)
3.12 条件表达式运算符 .....	(53)
复习题 .....	(54)
习题 .....	(55)
复习题答案 .....	(56)
习题答案 .....	(60)
<b>第4章 迭代</b> .....	(66)
4.1 while 语句 .....	(66)
4.2 循环的终止 .....	(68)
4.3 do..while 循环 .....	(70)
4.4 for 语句 .....	(72)
4.5 break 语句 .....	(78)
4.6 continue 语句 .....	(80)
4.7 goto 语句 .....	(80)
4.8 生成伪随机数 .....	(82)
复习题 .....	(87)
习题 .....	(88)
复习题答案 .....	(89)
习题答案 .....	(90)
<b>第5章 函数</b> .....	(95)
5.1 介绍 .....	(95)
5.2 标准 C++ 的库函数 .....	(95)

---

5.3 用户自定义函数	(98)
5.4 测试程序	(99)
5.5 函数声明和定义	(101)
5.6 局部变量和函数	(103)
5.7 void 函数	(105)
5.8 布尔函数	(106)
5.9 I/O 函数	(110)
5.10 引用传递	(111)
5.11 通过常量引用传递	(115)
5.12 内联函数	(116)
5.13 作用域	(117)
5.14 重载	(118)
5.15 main () 函数	(119)
5.16 默认的参数	(120)
复习题	(121)
习题	(121)
复习题答案	(124)
问题答案	(125)
<b>第 6 章 数组</b>	(137)
6.1 介绍	(137)
6.2 数组的处理	(137)
6.3 数组的初始化	(139)
6.4 数组元素下标越界	(141)
6.5 将数组传递给函数	(142)
6.6 线性查找算法	(145)
6.7 冒泡排序算法	(145)
6.8 二分查找算法	(146)
6.9 使用枚举类型的数组	(149)
6.10 类型定义	(150)
6.11 多维数组	(151)
复习题	(154)
习题	(155)
复习题答案	(160)
习题答案	(160)
<b>第 7 章 指针和引用</b>	(171)
7.1 引用运算符	(171)

7.2  引用 .....	(172)
7.3  指针 .....	(174)
7.4  取值操作符 .....	(175)
7.5  派生类型 .....	(177)
7.6  对象和左值 .....	(177)
7.7  返回引用 .....	(178)
7.8  数组和指针 .....	(179)
7.9  动态数组 .....	(184)
7.10  通过指针使用 const .....	(185)
7.11  指针数组和指向数组的指针 .....	(186)
7.12  指向指针的指针 .....	(187)
7.13  指向函数的指针 .....	(187)
7.14  NUL、NULL 和 void .....	(188)
复习题 .....	(189)
习题 .....	(192)
复习题答案 .....	(195)
习题答案 .....	(197)
<b>第 8 章 字符串</b> .....	(202)
8.1  介绍 .....	(202)
8.2  复习指针 .....	(202)
8.3  字符串 .....	(204)
8.4  字符串的输入输出 .....	(205)
8.5  cin 成员函数 .....	(206)
8.6  标准的 C 字符函数 .....	(209)
8.7  字符串数组 .....	(210)
8.8  标准 C 的字符串函数 .....	(212)
复习题 .....	(219)
习题 .....	(222)
复习题答案 .....	(223)
习题答案 .....	(224)
<b>第 9 章 标准 C++ 字符串</b> .....	(233)
9.1  引言 .....	(233)
9.2  格式化的输入 .....	(233)
9.3  非格式化输入 .....	(235)
9.4  标准 C++ 字符串类型 .....	(236)
9.5  文件 .....	(238)

9.6 字符串流 .....	(240)
复习题 .....	(241)
习题 .....	(242)
复习题答案 .....	(246)
习题答案 .....	(247)
<b>第 10 章 类</b> .....	(254)
10.1 引言 .....	(254)
10.2 类的声明 .....	(254)
10.3 构造函数 .....	(257)
10.4 构造函数初始化列表 .....	(259)
10.5 访问函数 .....	(260)
10.6 私有成员函数 .....	(260)
10.7 复制构造函数 .....	(262)
10.8 类的析构函数 .....	(264)
10.9 常量对象 .....	(265)
10.10 结构体 .....	(265)
10.11 对象的指针 .....	(266)
10.12 静态数据成员 .....	(267)
10.13 静态函数成员 .....	(269)
复习题 .....	(271)
习题 .....	(271)
复习题答案 .....	(272)
习题答案 .....	(273)
<b>第 11 章 重载运算符</b> .....	(279)
11.1 引言 .....	(279)
11.2 赋值运算符重载 .....	(279)
11.3 this 指针 .....	(280)
11.4 算术运算符重载 .....	(281)
11.5 算术赋值运算符重载 .....	(283)
11.6 关系运算符重载 .....	(283)
11.7 字符串运算符重载 .....	(284)
11.8 运算符转换 .....	(286)
11.9 加减运算符重载 .....	(287)
11.10 下标运算符重载 .....	(289)
复习题 .....	(290)
习题 .....	(291)

复习题答案	(291)
习题答案	(292)
<b>第 12 章 组合和继承</b>	(297)
12.1 引言	(297)
12.2 组合	(297)
12.3 继承	(299)
12.4 保护型类成员	(300)
12.5 重载和操纵继承成员	(303)
12.6 私有访问对保护访问	(306)
12.7 虚函数和多态性	(306)
12.8 虚拟析构函数	(309)
12.9 抽象基类	(311)
12.10 面向对象程序设计	(314)
复习题	(316)
习题	(316)
复习题答案	(317)
习题答案	(318)
<b>第 13 章 模板与迭代符</b>	(326)
13.1 引言	(326)
13.2 函数模板	(326)
13.3 类模板	(328)
13.4 容器类	(331)
13.5 子类模板	(332)
13.6 把模板类传到模板参数	(334)
13.7 链表的一个类模板	(336)
13.8 循环类	(339)
复习题	(346)
习题	(346)
复习题答案	(346)
习题答案	(347)
<b>第 14 章 标准 C++ 向量</b>	(352)
14.1 引言	(352)
14.2 关于向量的迭代符	(354)
14.3 赋值向量	(355)
14.4 erase() 和 insert() 函数	(356)
14.5 find() 函数	(358)

14.6 C++ 标准向量类模板 .....	(359)
14.7 范围检查 .....	(361)
复习题 .....	(361)
习题 .....	(361)
复习题答案 .....	(362)
习题答案 .....	(363)
<b>第 15 章 容器类</b> .....	(367)
15.1 ANSI/ISO 标准 C++ .....	(367)
15.2 标准模板库 .....	(367)
15.3 标准 C++ 容器类模板 .....	(367)
15.4 标准 C++ 的一般算法 .....	(368)
15.5 头文件 .....	(369)
<b>附录 A 字符代码</b> .....	(371)
A.1 ASCII 码 .....	(371)
A.2 Unicode .....	(374)
<b>附录 B 标准 C++ 关键字</b> .....	(377)
<b>附录 C 标准 C++ 运算符</b> .....	(379)
<b>附录 D 标准 C++ 容器类</b> .....	(381)
D.1 vector 类模板 .....	(381)
D.2 deque 类模板 .....	(386)
D.3 stack 类模板 .....	(387)
D.4 queue 类模板 .....	(387)
D.5 priority_queue 类模板 .....	(387)
D.6 list 类模板 .....	(389)
D.7 map 类模板 .....	(391)
D.8 set 类模板 .....	(393)
<b>附录 E 标准 C++ 一般算法</b> .....	(395)
<b>附录 F 标准 C 库</b> .....	(425)
<b>附录 G 十六进制数</b> .....	(430)



# 第 1 章 C++ 程序设计基础

程序就是能够由计算机执行的一组指令序列，所有的程序都由某种程序设计语言写成。C++ 是功能最强大的程序设计语言之一，程序设计人员使用它能够写出高效、结构化及面向对象的程序

## 1.1 入门

为了编写及运行 C++ 程序，在计算机中必须已经安装有文本编辑器及 C++ 编译器。文本编辑器是一种可以在计算机上创建及编辑文本文件的软件，程序设计人员使用它编写某种程序设计语言的程序。编译器是一种将程序转化成操作系统能够执行的机器语言（称为二进制编码）的一种软件。这种转化过程称做“编译”。一个 C++ 编译器可以将 C++ 程序编译成机器语言。

如果计算机正运行着微软公司 Windows 操作系统的一个版本（如 Windows 98 或 Windows 2000），则其中已经带有两个文本编辑器：写字板和记事本。可以通过“开始”按钮来启动。在 Windows 98 系统中，它们在“附件”里。

Windows 操作系统没有内建的 C++ 编译器，所以除非已经有人在机器上安装过 C++ 编译器，否则必须自己安装。如果使用的是由别人管理的运行 Windows 系统的计算机（比如公司或学校的信息服务部门），则 C++ 编译器可能已经安装好了。使用“开始”按钮，在“程序”目录下查找 Borland C++ Builder, Metrowerks CodeWarrior, Microsoft Visual C++ 或任何名字中包含 C++ 的程序即可。如果不得不自己购买 C++ 编译器，则可到网上查找所有上面提到的编译器中不太贵的版本。这些通常是指 IDE（集成开发环境），因为它们包含特定的文本编辑工具及调试工具。

如果使用的计算机是运行着个人版本 UNIX 操作系统的工作站（如运行 Sun Solaris 的 SPARC 工作站），它可能已经安装有 C++ 编译器。一个简便的寻找方法就是建立如例 1.1 所示的程序，将它命名为 hello.c，并用以下命令尝试编译它：

```
cc hello
```

自由软件基金有一套 UNIX 软件，称为“GNU”软件，它们可以从以下网址免费下载：

<http://www.gnu.org/software/software.html>

可以使用包含 C++ 编译器的 GCC 包和 Emacs 编辑器。在 DOS 系统下，可以使用包含 C++ 编译器的 DJGPP。

## 1.2 程序实例

假设已有一个文本编辑器来编写 C++ 程序, 也有一个 C++ 编译器来编译它们。若在人计算机上使用集成开发环境如 Borland C++ Builder, 则可以通过按一个合适的按钮来编译与运行程序。其他的系统可能会要求在命令行中运行程序。如果是这样的话, 则可像键入命令一样键入文件名。例如, 若源代码在名为 hello.cpp 的文件中, 则键入:

```
hello
```

就可以在编译完成之后运行这个程序。

在编写 C++ 程序时, 必须注意 C++ 是大小写敏感的, 也就是说 main () 不同于 Main ()。最安全的方法是除非必须大写, 否则全部采用小写。

### 例 1.1 “Hello, World” 程序

本例简单地打印出 “Hello, World!”:

```
#include <iostream>
int main ()
{ std::cout << "Hello, World! \n";
}
```

代码的第 1 行是一条预处理伪指令, 它告诉 C++ 编译器在什么地方寻找第 3 行中使用的 std::cout 对象的定义。标识符 iostream 是标准 C++ 库中一个文件的名称。所有用到标准输入输出的 C++ 程序都必须包含这个预处理伪指令。注意所需的符号: 必须用井号 # 指明 include 是一个预处理伪指令; 用符号 < > 指明 iostream (它代表输入输出流) 是一个标准 C++ 库文件的名称。表达式 <iostream> 称做标准头。

第 2 行也是所有 C++ 程序中必须有的。它声明了程序在何处开始。标识符 main 是一个函数的名称, 称为程序的主函数。所有的 C++ 程序都必须有且仅有一个 main () 函数。main 后面所跟的圆括号表明这是一个函数。关键词 int 是 C++ 中的一种数据类型, 它代表整数。在这里使用它表明 main () 函数返回值的类型。当程序结束运行时, 它可以向操作系统返回一个表示某种结果状态的整数类型值。

最后两行组成了程序的体, 程序体是包含在花括号 “{}” 中的一系列程序语句。在本例中只有一条语句:

```
std::cout << "Hello, World! \n";
```

它说明将字符串 “Hello, World! \n” 发送给标准输出流 std::cout 对象。符号 “<<” 代表 C++ 的输出运算符。当这条语句被执行时, 包在引号 “” 中的字符被送至标准输出设备, 一般是计算机屏幕。最后两个字符 \n 代表换行符。当输出设备遇到这个字符时, 它向前移动至屏幕上文件下一行的开始处。最后, 注意所有的程序语句都必须以分号 “;” 结尾。

注意，例 1.1 的程序是如何在四行中进行安排的，这种格式有助于阅读。C++ 编译器忽略这些格式，它所读到的内容就如程序全部写在一行中，如下所示：

```
#include <iostream>
int main () { std::cout << "Hello, World! \n"; }
```

编译器会忽略空格符，除非它们被用于分开标识符如：

```
int main
```

注意预编译命令必须在程序之前的单独一行中。

## 例 1.2 另一个“Hello, World”程序

本例输出同例 1.1：

```
#include <iostream>
using namespace std;
int main ()
{ // 打印 "Hello, World!":
  cout << "Hello, World! \n";
  return 0;
}
```

第 2 行：

```
using namespace std;
```

告诉 C++ 编译器在需要使用前缀的地方使用 `std::`。它表明可以用 `cout` 代替 `std::cout`。这使大型程序比较容易阅读。

第 4 行：

```
    // 打印 "Hello, World!":
```

包含注释“打印“Hello, World!”:”。程序中的注释是在程序编译之前被预处理去掉的一串字符，它们被用于向阅读者提供解释。在 C++ 中，所有从双斜杠“//”至行尾之间的文字被视为注释。也可以像下面一样使用 C 语言风格的注释：

```
    /* 打印 "Hello, World!": */
```

C 语言风格的注释（由程序设计语言“C”引入）是指在符号“/\*”与“\*/”间的任何字符串，这些字符串可以跨过多行。

第 6 行：

```
    return 0;
```

在标准 C++ 的 `main ()` 中是可选的。在这里包含这行只是因为有一些编译器希望 `main ()` 函数在最后一行包含它。

名字空间是一个被命名的定义集合。当一个名字空间中定义的对象被用于此名字空间以

外时，或者他们以所属名字空间为前缀，或者必须在采用“using namespace”语句预先定义的块中。名字空间允许程序对不同的对象使用同一个名字，就像不同的人可以有相同的名字一样。cout 对象是在 <iostream> 头文件中 std（代表标准）名字空间中定义的。

在本书的其余部分，所有的程序都假定以下面两行开始：

```
#include <iostream>
using namespace std;
```

在例子中这两必要的两行将被省略，并且也将在 main（）函数中省略：

```
return 0;
```

如果使用要求这行语句的编译器（如 Microsoft Visual C++），则应将它包含进程序。

### 1.3 输出运算符

符号“<<”在 C++ 中被称为输出运算符（也被称为 put 运算符或流插入运算符）。它将值插入到左边的输出流中。经常会使用 cout 输出流，它通常指向计算机屏幕。所以语句：

```
cout << 66;
```

将在屏幕上显示数字 66。

运算符将动作作用于一个或多个对象。输出运算符 << 完成将其右边表达式的值送到左边流的任务。由于这个动作的方向看起来是从右向左的，因此选用 << 来表示。它用指向左边的箭头进行提醒。

对象 cout 被称为一个“流”，因为传送给它的输出数据象水流一样流动。如果有多个数据被插入到 cout 流中，它们就像落入水中一样一个接一个排成一条线。这就像树叶从树上落入真正的水流中。插入 cout 流中的数据按照这样的顺序被显示在屏幕上。

#### 例 1.3 又一个“Hello, World”程序

本例输出同例 1.1：

```
int main ()
{ // 打印 "Hello, World!":
  cout << "Hel" << "lo, Wo" << "rld!" << endl;
}
```

这里输出运算符被使用了四次，按顺序将四个对象“Hel”、“lo, Wo”、“rld!”及“endl”放入输出流。前三个对象是字符串，它们被连接（头尾相连排成一列）在一起组成一个单个字符串“Hello, World”。第四个对象是流操作对象 endl（意思是“行尾”），它的作用相当于在字符串后添加行尾符‘\n’；将打印光标送到下一行的开始处，并且清空缓冲区。

## 1.4 字符与文字

例 1.3 中的“Hel”、“lo, Wo”及“rld!”三个对象被称为“字符串文字”。每个文字由一组被引号界定的字符序列组成。

字符是记录有意义书写的基本符号。英语作家使用拉丁字母表中的 26 个小写字母和 26 个大写字母, 同时还有 10 个阿拉伯数字及其他一些符号。字符在计算机中是以整数形式存储的。字符集码就是列出字符集中每个字符的整数表示值的表。在 2000 年前使用最广泛的字符集是 ASCII 码, 见附录 A 中。它是美国信息交换标准码的首字母缩写 (读作 “as-key”)。

换行符“\n”是不可打印字符之一。它是由反斜杠“\”与字母 n 组成的单个字符。其他还有一些字符也采用相同的方法组成, 包括制表符“\t”及报警符“\a”。反斜杠还用于指示两个无法在文字中使用的可打印字符, 引号“\”及反斜杠自身“\\”。

字符在程序语句中可以作为字符串文字的一部分, 也可以作为单独的对象。当单独使用时, 必须作为字符常量。字符常量是由单引号括起来的单个字符。作为单独的对象, 字符常量可以像字符串文字一样被输出。

### 例 1.4 “Hello, World” 程序的第四个版本

本例输出同例 1.1:

```
int main ()
{ // 打印 "Hello, World!";
  cout << "Hello, W" << 'o' << "rld" << '!' << '\n';
}
```

这个程序表明输出运算符可以像处理字符串文字一样很好的处理字符。三个单独的字符 ‘o’, ‘!’ 及 ‘\n’ 以与两个字符串文字 “Hello, W” 及 “rld” 相同的方式被连接到输出上。

### 例 1.5 在标准输出流中插入数字文字

```
int main ()
{ // 打印 "The Millennium ends Dec 31 2000.":
  cout << "The Millennium ends Dec " << 3 << 1 << ' ' << 2000 << endl;
}
```

当 3 和 2000 这样的数字文字被送至输出流时, 它们被自动转换成字符串文字, 并以与字符相同的方式被连接至输出流。注意, 为了不使数字靠在一起, 必须明确地传送空格字符 ‘ ’。

## 1.5 变量及其他声明

变量是代表计算机内存中一个存储位置的符号。存储在此位置的信息被称做这个变量的值。变量得到值的通用方法是赋值, 语法形式是:

变量 = 表达式;

表达式先被求值并将结果赋值给变量。等号“=”在C++中是赋值运算符。

### 例 1.6 使用整型变量

本例中，整数 44 被赋值给变量 m，表达式 m + 33 被赋值给变量 n：

```
int main ()
{ // 打印 "m = 44 and n = 77":
  int m, n;
  m = 44; // 将值 44 赋值给变量 m
  cout << "m = " << m;
  n = m + 33; // 将值 77 赋值给变量 n
  cout << " and n = " << n << endl;
}
```

这个程序的输出在以下的阴影框中：

m = 44 and n = 77

可以这样看变量 m 和 n：

变量 m 就像一个邮箱，它的名称 m 像邮箱上的地址；它的值 44 像邮箱内的物品；它的类型 int 像合法性检查，以确保只有哪些东西可以放入，类型 int 表明这个变量只拥有整数值。



注意在这个例子中，变量 m 和 n 是在一行中声明的。具有相同类型的任意多个变量都可以像这样声明在一行之中。

C++ 中所有变量在使用前都必须声明，语法是：

```
specifier type name initializer;
```

specifier 是一个可选的关键字，如 constant；type 是 C++ 的数据类型之一，如 int；name 是变量的名称；initializer 是可选的初始化子句，如 “= 44”（见 1.7 节）。

声明变量的作用是在程序中引入一个名称，也就是告诉编译器一个名称的意思。类型 type 告诉编译器这个变量值的范围以及可以对它进行哪些操作。

变量在程序中声明的位置决定了这个变量的作用域：即变量可以在程序中的哪部分使用。通常，变量的作用域是从它被声明处到包含它的声明及操作的直接块的结束。

## 1.6 程序标记

计算机程序是由一系列称为“标记”的单元组成。这些标记包括关键字如“int”；标识符如“main”；符号如“{”；运算符如“<<”。当编译程序时，编译器将检查源代码，将它分析成标记。如果它发现意外的标记或未发现期望的标记，它将中断编译并给出错误信息。比如，如果忘记在一行的结尾加上分号，则错误信息会指出缺少分号。一些语法错误，如缺

少第二个引号或缺少结束的花括号一般无法被准确指出，编译器只能指出在程序的某处附近发现了错误。

### 例 1.7 程序的标记

```
int main ()
{ // 打印 "n = 44":
  int n = 44;
  cout << "n = " << n << endl;
}
```

这个程序的输出是：

```
n=44
```

源代码中包含了 19 个标记：“int”、“main”、“(”、“)”、“|”、“int”、“n”、“=”、“44”、“;”、“cout”、“<<”、“n = ”、“<<”、“n”、“<<”、“endl”、“;”及“|”。注意，编译器将忽略注释符//及这行中其后的文字。

### 例 1.8 一个有错的例子

本例与上例相同，除了第三行中缺少所需的分号：

```
int main ()
{ // 此代码有错误
  int n = 44
  cout << "n = " << n << endl;
}
```

一种编译器会如下错误信息：

```
Error :      ';' expected
Testing.cpp line 4      cout << "n = " << n << endl;
```

这个编译器将给所发现的错误标记加下划线，如在这个例子中，是第四行最前面的“cout”。只有在检查到下一个标记的时候，遗漏的标记才会被发现。

## 1.7 初始化变量

在大多数情况下，声明变量的同时进行初始化是明智的方法。

### 例 1.9 初始化变量

本例包含一个初始化的变量及一个未初始化的变量。

```
int main ()
{ // 打印 "m = ?? and n = 44":
  int m;    // BAD: m 未被初始化
  int n = 44;
```

```
int n = 44;
cout << "m = " << m << " and n = " << n << endl;
|
m = ?? and n = 44
```

输出如上阴影框所示。

这个编译器以一种特殊的方式处理未初始化的变量。它赋给它们一个特殊的值，使它们在打印时显示“??”。其他的编译器可能只是简单地将“垃圾”数据赋给变量，使得输出时显示：

```
m = -2107339024 and n = 44
```

在大型程序中，未初始化的变量可能会引起麻烦的错误。

## 1.8 对象、变量和常量

对象就是内存中相连的一段区域，它有地址、大小、类型和值。对象的地址就是内存中这个对象第一个字节的地址。对象的大小就是它占内存的字节数。对象的值就是它在内存中实际存放的位以及由对象的类型所决定的这些位如何被解释成数据。

例如，在 UNIX 工作站上的 GNU C++ 中，对象 *n* 被定义为：

```
int n = 22;
```

它的地址为 0x3ffcd6，大小为 4，类型为 int，值为 22（内存地址为十六进制数，见附录 G）。

对象的类型是由程序设计人决定的。它的值可能也是由程序设计人员在编译时决定的，或是在运行时决定的。对象的大小是由编译器所决定的。比如，在 GNU C++ 中，int 值的大小是 4，而在 Borland C++ 中，它的大小是 2。对象的地址是在运行时由计算机的操作系统决定的。

有一些对象没有名字。一个变量就是一个有名字的对象。上面定义的对象就是名字为 *n* 的变量。

“变量”这个词暗示这个对象的值是可以改变的。一个值不能改变的对象被称为常量。常量的定义是在类型标识符之前加上关键词 *const*。如：

```
const int N = 22;
```

常量在声明时必须被初始化。

### 例 1.10 const 标识符

这个例子说明了常量的定义：

```
int main ()
{ // 定义常量，无输出
```



```

const char BEEP = '\b';
const int MAXINT = 2147483647;
const int N = MAXINT/2;
const float KM_PER_MI = 1.60934;
const double PI = 3.14159265358979323846;

```

常量通常用来定义像  $\pi$  这类将在程序中多次使用且不会改变的值。

通常习惯用全部大写字母组成常量的标识符，以使它们与其他类型的标识符相区别。一个好的编译器将会用常量的数字值来替换每个常量符号。

## 1.9 输入运算符

在 C++ 中，输入几乎与输出一样简单。输入运算符 `>>`（也称做 `get` 运算符或取出运算符）和输出运算符 `<<` 工作方式一样。

### 例 1.11 使用输入运算符

```

int main ()
{ // 检查输入的整数、浮点数和字符
    int m, n;
    cout << "Enter two integers: ";
    cin >> m >> n;
    cout << "m = " << m << ", n = " << n << endl;
    double x, y, z;
    cout << "Enter three decimal numbers: ";
    cin >> x >> y >> z;
    cout << "x = " << x << ", y = " << y << ", z = " << z << endl;
    char c1, c2, c3, c4;
    cout << "Enter four characters: ";
    cin >> c1 >> c2 >> c3 >> c4;
    cout << "c1 = " << c1 << ", c2 = " << c2 << ", c3 = " << c3
        << ", c4 = " << c4 << endl;
}

```

```

Enter two integers: 22 24
m = 22, n = 44
Enter three decimal numbers: 2.2 4.4 6.6
x = 2.2, y = 4.4, z = 6.6
Enter four characters: ABCD
c1 = A, c2 = B, c3 = C, c4 = D

```

本例输入部分用粗体标出。

## 复 习 题

1.1 说明 C++ 程序中两种包含注释的方法。

## 1.2 以下程序有何错误?

```
#include <iostream>
int main ()
{ // 打印 "Hello, World!";
  cout << "Hello, World! \n";
}
```

## 1.3 以下的 C 语言风格注释有何错误?

```
cout << "hello, /* change? */ World. \n";
```

## 1.4 以下程序有何错误?

```
#include <iostream>;
int main
{ // 打印 "n = 22";
  n = 22;
  cout << "n = << n << endl;
}
```

## 1.5 声明的作用是什么?

## 1.6 预处理伪指令的作用是什么?

```
#include <iostream>
```

## 1.7 最短的 C++ 程序可能是什么?

## 1.8 “C++” 的名字是从何而来?

## 1.9 以下这些声明有何错误:

```
int first = 22, last = 99, new = 44, old = 66;
```

1.10 在以下各语句中, 假设在语句执行前  $m$  的值为 5,  $n$  的值为 2。请说出各语句执行后  $m$  与  $n$  的值。

a.  $m * = n++$ ;

b.  $m + = --n$ ;

1.11 假设  $m$  的值为 25,  $n$  的值为 7, 请给出下列表达式的值:

a.  $m - 8 - n$

b.  $m = n = 3$

c.  $m * n$

d.  $m \% n++$

e.  $m \% ++n$

f.  $++m - n--$

## 1.12 分析以下程序, 找出所有的关键词、标识符、运算符、文字、符号及注释:

```
int main ()
{ int n;
```

```

    cin >> n;
    n *= 3;    // n 乘 3
    cout << "n=" << n << endl;
}

```

1.13 找出并改正下列各句中的错误:

a. `cout >> count`

b. `int double = 44;`

1.14 以下两条语句有何不同:

```

char ch = 'A';
char ch = 65;

```

1.15 用何方法可以找出 ASCII 码为 100 的字符?

1.16 “浮点数”是何意思,为什么这么叫?

1.17 什么是数值溢出?

1.18 整数溢出与浮点数溢出有何不同?

1.19 什么是运行时错误?请给出两种不同类型的运行错误的例子。

1.20 什么是编译错误?请给出两种不同的类型的编译错误的例子。

## 习 题

1.1 写出 4 条不同的语句,每条都从整数型变量 `n` 中减去 1。

1.2 写出与以下语句功能相同但不使用后增量运算符 (`++`) 的一段 C++ 代码:

```
n = 100 + m++;
```

1.3 写出与以下语句功能相同但不使用前增量运算符 (`++`) 的一段 C++ 代码:

```
n = 100 + ++m;
```

1.4 写出一条 C++ 语句,实现将 `z` 减去 `x` 与 `y` 的和并加到 `y` 上。

1.5 写出一条 C++ 语句,实现将变量 `n` 减 1 后加到 `total` 里。

1.6 写出一个能打印 Gettysburg Address (或其他你喜欢的引用) 第一句的程序。

1.7 写出一个程序,实现在  $7 \times 6$  网格中用星号打印如下的印刷体字母 “B”:

```

* * * * *
*           *
*           *
* * * * *
*           *
*           *
* * * * *

```

- 1.8 编写并运行一个程序，实现在  $7 \times 7$  网格中用星号打印你的姓中的第一个字母。
- 1.9 编写并运行一个程序，显示下列 10 个转义符被打印时的情况：\a, \b, \n, \r, \t, \v, \', \", \\, \?
- 1.10 编写并运行一个程序，打印出两个整数 60 和 7 的和、差、积、商及余数。
- 1.11 编写并运行一个程序，打印出两个通过交互方式输入的整数的和、差、积、商及余数。
- 1.12 编写并试运行一个程序，显示你的系统是如何处理未经初始化的变量。
- 1.13 编写并运行一个能引起 short 变量负溢出的程序。
- 1.14 按以下步骤编写并运行一个演示四舍五入错误的程序：  
 (1) 初始化浮点数变量 a 为 666666；(2) 初始化浮点数变量 b 为  $1 - 1/a$ ；(3) 初始化浮点数变量 c 为  $1/b - 1$ ；(4) 初始化浮点数变量 d 为  $1/c + 1$ ；(5) 打印出这 4 个变量。在数学上  $d = a$ ，但计算的值  $d \neq a$ ，这是由四舍五入的错误引起的。

## 复习题答案

- 1.1 一种方法是使用 C 语言风格注释

```
/* 像下面这样 */
```

另一种方法是使用 C++ 语言风格注释：

```
// 像下面这样
```

第一种从斜框加星号开始，到星号加斜框结束。第二种从双斜框开始，到这行结束为止。

- 1.2 最后一条语句缺少分号。
- 1.3 双引号中所有的内容都将被打印，包括要想注释的内容。
- 1.4 有 4 个错误：第一行的预编译伪指令不应该以分号结尾；main() 函数缺少圆括号；n 没有声明；最后一行缺少结束的引号。
- 1.5 声明告诉编译器所声明的变量名及类型，也可以在声明时进行初始化。
- 1.6 它将头文件 iostream 的内容包含进程序，这个包含声明对于输入、输出是必须的，例如：输出运算符 <<。
- 1.7 `int main() {}`
- 1.8 这个名字涉及了 C 语言和它的增量运算符“++”。这暗示 C++ 比 C 更先进。
- 1.9 这个声明惟一的错误在于 new 是关键字。关键字被保留，不能用于变量的名。C++ 中的 62 个关键字见附录 B。
- 1.10 a. m 是 10, n 是 3  
 b. m 是 6, n 是 1
- 1.11  $a \cdot m - 8 - n$  结果为  $(25 - 8) - 7 = 17 - 7 = 10$

- b. m = n = 3* 结果为 3
- 1.12 *a. m = 8 - n* 结果为  $(25 - 8) - 7 = 17 - 7 = 10$   
*b. m = n = 3* 结果为 3  
*c. m % n* 结果为  $25 \% 7 = 4$   
*d. m % n++* 结果为  $25 \% (7++) = 25 \% 7 = 4$   
*e. m % ++n* 结果为  $25 \% (++7) = 25 \% 8 = 1$  (应该是 3)  
*f. ++m - n--* 结果为  $(++25) - (7--) = 26 - 7 = 19$
- 1.13 关键字有 `int`。标识符有 `main`、`n`、`cin`、`cout` 和 `endl`。运算符有 `()`、`>>`、`*=` 和 `<<`。文字有 3 和 “n=”。符号有 `{`、`;` 和 `}`。注释有 “// n 乘以 3”。
- 1.14 *a.* 输出对象 `cout` 需要输出运算符 `<<`，应该是 `cout << count`；  
*b.* `double` 是 C++ 的关键字，它不能做变量的名，可以用 `int d = 44`；
- 1.15 两条语句功能相同：它们将 `ch` 声明为 `char` 型，并初始化为 65。由于这正好是 A 的 ASCII 码，所以它也可以用于初始化这个字符。
- 1.16 `cout << "char (100) =" << char (100) << endl`；
- 1.17 “浮点”这个术语用于说明小数（有理数）在计算机内存存储的方式。这个名字说明像 386 501.294 这样的小数可以通过将小数点向左“浮动”5 位表示成  $3.8650129 \times 10^5$ 。
- 1.18 程序中当一个数值变量的值对于它的类型来说太大时，会发生数值溢出。比如：在多数计算上，`short` 型的变量不能超过 32 767，因此当一个这种类型的变量值为 32 767 时，如果再被加（或通过任何算法操作被加），则会发生溢出。
- 1.19 当发生整数溢出时，这个变量会被“反卷”成负数，产生错误的结果。当发生浮点数溢出时，这个变量的值会被设为代表无穷大的常数 `inf`。
- 1.20 运行时错误发生在程序运行的时候，数值溢出及除 0 都是运行时错误的例子。
- 1.21 编译错误发生在程序被编译的时候，比如：缺少所需分号的语法错误；使用了未声明的变量；使用关键字作为变量名。

## 习题答案

- 1.1 将变量 `n` 减 1 的 4 条语句

*a.* `n = n - 1`;

*b.* `n--`;

*c.* `--n`;

*d.* `n--`;

- 1.2 `n = 100 + m`;

`++m`;

- 1.3 `++m`;

`n = 100 + m`;



```

1.9 int main()
{ // 打印转义字符
  cout << "Prints \"\\nXXYY\": " << "\\nXXYY" << endl;
  cout << "-----" << endl;
  cout << "Prints \"\\nXX\\bYY\": " << "\\nXX\\bYY" << endl;
  cout << "-----" << endl;
  cout << "Prints \"\\n\\tXX\\tYY\": " << "\\n\\tXX\\tYY" << endl;
  cout << "-----" << endl;
  cout << "Prints the '\\a' character: " << "\\a" << endl;
  cout << "-----" << endl;
  cout << "Prints the '\\r' character: " << "\\r" << endl;
  cout << "-----" << endl;
  cout << "Prints the '\\v' character: " << "\\v" << endl;
  cout << "-----" << endl;
  cout << "Prints the '\\?' character: " << "\\?" << endl;
  cout << "-----" << endl;
}

```

```

Prints the '\\v' character:
-----
Prints the '\\?' character: ?
-----
Prints "\\nXXYY":
XXYY
-----
Prints "\\nXX\\bYY":
XXY
-----
Prints "\\n\\tXX\\tYY":
  XX    YY
Prints the '\\a' character:
-----
Prints the '\\r' character:
-----

```

```

1.10 int main()
{ // 打印数学运算符的结果
  int m = 60, n = 7;
  cout << "The integers are " << m << " and " << n << endl;
  cout << "Their sum is      " << (m + n) << endl;
  cout << "Their difference is " << (m - n) << endl;
  cout << "Their product is    " << (m * n) << endl;
  cout << "Their quotient is   " << (m / n) << endl;
  cout << "Their remainder is  " << (m % n) << endl;
}

```

```

The integers are 60 and 7
Their sum is      67
Their difference is 53
Their difference is 53
Their product is  420
Their quotient is  8
Their remainder is 4

```

```

1.11 int main()
{ // 打印数学运算符的结果
    int m, n;
    cout << "Enter two integers: ";
    cin >> m >> n;
    cout << "The integers are " << m << " and " << n << endl;
    cout << "Their sum is " << (m + n) << endl;
    cout << "Their difference is " << (m - n) << endl;
    cout << "Their product is " << (m * n) << endl;
    cout << "Their quotient is " << (m / n) << endl;
    cout << "Their remainder is " << (m % n) << endl;
}

```

```

Enter two integers: 60 7
The integers are 60 and 7
Their sum is 67
Their difference is 53
Their product is 420
Their quotient is 8
Their remainder is 4

```

```

1.12 int main()
{ // 打印未初始化变量
    bool b; // 未初始化
    cout << "b = " << b << endl;
    char c; // 未初始化
    cout << "c = [" << c << "]" << endl;
    int m; // 未初始化
    cout << "m = " << m << endl;
    int n; // 未初始化
    cout << "n = " << n << endl;
    long nn; // 未初始化
    cout << "nn = " << nn << endl;
    float x; // 未初始化
    cout << "x = " << x << endl;
    double y; // 未初始化
    cout << "y = " << y << endl;
}

```

```

b = 0
c =
m = 4296913
n = 4296716
nn = 4296794
x = 6.02438e-39
y = 9.7869e-307

```

```

1.13 int main()
{ // 打印溢出的负短整型的值
    short m=0;
    cout << "m = " << m << endl;
    m -= 10000; // m 应为 -10.000
    cout << "m = " << m << endl;
    m -= 10000; // m 应为 -20.000
}

```



```
cout << "m = " << m << endl;
m -= 10000; // m 应为 -30.000
cout << "m = " << m << endl;
m -= 10000; // m 应为 -40.000
cout << "m = " << m << endl;
}
```

```
m = 0
m = -10000
m = -20000
m = -30000
m = 25536
```

```
1.14 int main()
{ float a = 666666;           // = a = 666666
  float b = 1 - 1/a;          // = (a-1)/a = 666665/666666
  float c = 1/b - 1;          // = 1/(a-1) = 1/666665
  float d = 1/c + 1;          // = a = 666666 != 671089
  cout << "a = " << a << endl;
  cout << "b = " << b << endl;
  cout << "c = " << c << endl;
  cout << "d = " << d << endl;
}
```

```
a = 666666
b = 0.999999
c = 1.49012e-06
d = 671089
```

## 第2章 基本类型

### 2.1 数值数据类型

科学研究中有两种类型的数字：整数（如 666）和小数（如 3.14159）。整数包括 0 及负整数，被称为整型。小数包括负小数及所有整数，它们都可以被表示成两个整数的比值（也就是分数），被称为有理数。数学中还使用实无理数（如 $\sqrt{2}$ 和 $\pi$ ），但是在计算机中它们必须被近似成有理数。

整数用于计数，有理数用于度量。整数表示精确，有理数表示近似。说陪审团中有 12 个人时，是确切地说有 12 个人，谁都可以数一数来验证。但若说树有 12m 高时，则通常指大约有 12.0m，另一个人也许会更精确地指出它实际是 12.01385m 高。

这种哲学上的二分法反映在计算机中，就是采用不同的方法存储与操作这两类本质不同数据。这种不同也体现在，所有程序设计语言中通常都包含两种数值类型：整型和浮点型。术语“浮点”是指用于记录小数的科学计数法。如，1234.56789 可以被表示成  $1.23456789 \times 10^3$ ，或 0.00098765 被表示成  $9.8765 \times 10^{-4}$ 。这种转换是将小数点在数字间“浮”动，并且使用 10 的指数形式记录向左或向右浮动的位数。

标准 C++ 有 14 种不同的基本类型：

11 种整型和 3 种浮点型，如图 2.1 中所示。整型包括布尔型 bool；用 enum 关键字定义的是枚举型；3 种字符型和 6 种明确的整数型。3 种浮点型分别是 float、double 和 long double。最常用的基本类型是 bool、char、int 和 double。

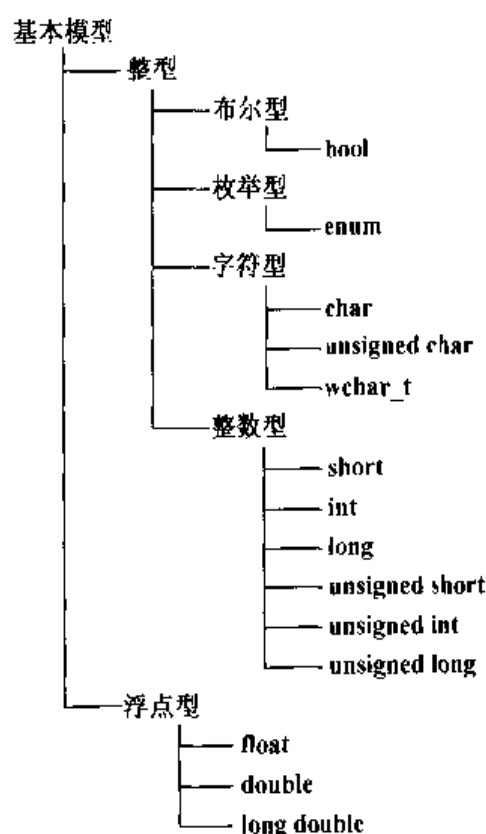


图 2.1 基本类型

## 2.2 布尔型

布尔型是一种整数型，但它的变量只能取两个值：false 和 true。这两个值被存为 0 和 1。标准 C++ 中，布尔型被称为 bool。

### 例 2.1 布尔类型变量

```
int main ()
| // 打印布尔变量的值
  bool flag = false;
  cout << "flag = " << flag << endl;
  flag = true;
  cout << "flag = " << flag << endl;
|
flag = 0
flag = 1
```

注意 false 值被打印成整数 0 而 true 值被打印成整数 1。

## 2.3 枚举型

除了预先定义的类型如 int 和 char，C++ 还允许自己定义特殊数据类型，这可以通过多种方式实现，其中最强大的方法是采用第 11 章中介绍的类。本节只考虑一种简单得多的自定义数据类型。

枚举型是一种整数型，用户通过以下语法定义：

```
enum typename { enumerator-list };
```

enum 是 C++ 的关键字，typename 是所定义类型的名字，enumerator-list 是一系列整数常量的名字。如，下面定义了枚举型 Semester，这种类型的变量可以有 3 个值。

```
enum Semester { FALL, SPRING, SUMMER};
```

然后可以定义下面这种类型的变量：

```
Semester s1, s2;
```

并使用这些变量及预定义的类型。

```
s1 = SPRING;
s2 = FALL;
if (s1 == s2) cout << "Same semester." << endl;
```

enumerator-list 中实际定义的值被称为枚举值。事实上，它们就是普通的整数常量。如上面 Semester 中定义的枚举值 FALL、SPRING 和 SUMMER 也可以被这样定义：

```
const int FALL = 0;
const int SPRING = 1;
const int SUMMER = 2;
```

当类型被定义时, 会被自动赋值 0, 1, ...。这些缺省的值也可以在 `enumerator-list` 中修改:

```
enum Coin {PENNY = 1, NICKEL = 5, DIME = 10, QUARTER = 25};
```

如果只有部分枚举值被赋了整数值, 则其他的会被赋予相邻的值。如:

```
enum Month {JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
```

中的 12 个月会被自动赋值 1 到 12。

由于枚举值只是简单的整数值, 所以不同的枚举可以拥有相同的值:

```
enum Answer {NO = 0, FALSE = 0, YES = 1, TRUE = 1, OK = 1};
```

这将使下列程序:

```
int answer;
cin >> answer;
:
:
if (answer == YES) cout << "You said it was o.k." << endl;
```

可以像预计那样执行。如果变量 `answer` 的值为 1, 则条件成立, 产生输出。注意, 由于整数 1 在条件语句中即代表“真”, 因此这条语句也可以被写成:

```
if (answer == YES) cout << "You said it was o.k." << endl;
```

注意, 这里特意用了大写字母。许多程序员都遵守采用大写字母定义他们标识符的下列约定:

- (1) 只用大写字母定义常量。
- (2) 将用户定义的类型名称的第一个字母大写。
- (3) 其他地方使用小写字母。

使用这些约定可以更容易地、尤其在大型程序中区分常量、类型及变量。约定 (2) 还有助于将标准 C++ 定义的变量类型如 `float` 及 `string` 与用户定义的类型如 `Coin` 与 `Month` 相区别。

定义枚举类型通常是为了使程序更明晰, 也就是使人更容易阅读。这里有一些典型的例子:

```
enum Sex {FEMALE, MALE};
enum Day {SUN, MON, TUE, WED, THU, FRI, SAT};
enum Radix {BIN = 2, OCT = 8, DEC = 10, HEX = 16};
enum Rank {TWO = 2, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN,
           JACK, QUEEN, KING, ACE};
enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```

```
enum Roman {I=1, V=5, X=10, L=50, C=100, D=500, M=1000};
```

像这样的定义可以使程序更具可读性。但是，枚举型也不应被过多使用。每个枚举值都定义了一个新的标识符。如，上面的 Roman 就将 I、V、X、L、C、D 和 M 这 7 个标识符定义成了常量，因此，在它们所定义的范围内就不能再被用于其他的用途了。

注意，枚举值必须是合法的标识符。如下列定义就不合法。

```
enum Grade {F, D, C-, C+, B-, B, B+, A-, A}; // 错误的
```

因为，字符 + 及 - 不能被用于标识符。而且，上面定义的 Month 和 Radix 都定义了符号 OCT，所以，不能被用于同一段程序范围。

C++ 中的枚举型也可以是无名的，例如：

```
enum {I=1, V=5, X=10, L=50, C=100, D=500, M=1000};
```


这只是一个定义整型常量的方便方法。

## 2.4 字符型

字符型是一种整型，它的变量代表像 A 的字母或像 8 的数字。字符文字由单引号'限定。像所有整型一样，字符值也以整型的方式存储。

### 例 2.2 字符变量

```
int main ()
{ // 打印字符和已在内部存储的整型值
    char c = 'A';
    cout << "c = " << c << ", int (c) = " << int (c) << endl;
    c = 't';
    cout << "c = " << c << ", int (c) = " << int (c) << endl;
    c = '\t'; // tab 字符
    cout << "c = " << c << ", int (c) = " << int (c) << endl;
    c = '!';
    cout << "c = " << c << ", int (c) = " << int (c) << endl;
}
```



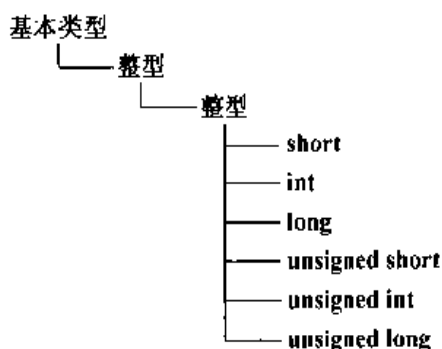
```
c = A, int (c) = 65
c = t, int (c) = 116
c = , int (c) = 9
c = !, int (c) = 33
```

由于字符值被用于输入和输出，因此它们表现为字符的形式而不是整型的形式：字符 A 是以字母 A 的形式而不是以它的内部表示的整数值 65 的形式被打印出来。这里使用类型转换运算符 int () 将其内部对应的整型值展示出来。这就是字母的 ASCII 码（见附录 A）。

## 2.5 整型值

标准 C++ 中有 6 种整型 (如图 2.2 所示): 这些类型实际上还有其他一些名字。如, short 也可以被称为 short int, int 也可以被称为 signed int。

可以通过运行如下程序确定系统中不同整型的数值范围。



### 例 2.3 整型范围

图 2.2 整型

```

#include <iostream>
#include <limits> // 定义常量
                  // SHRT_MIN 等等

int main ()
{ // 打印存储头文件 <limits> 中的一些常量
  cout << "minimum short = " << SHRT_MIN << endl;
  cout << "maximum short = " << SHRT_MAX << endl;
  cout << "maximum unsigned short = 0" << endl;
  cout << "maximum unsigned short = " << USHRT_MAX << endl;
  cout << "minimum int = " << INT_MIN << endl;
  cout << "maximum int = " << INT_MAX << endl;
  cout << "minimum unsigned int = 0" << endl;
  cout << "maximum unsigned int = " << UINT_MAX << endl;
  cout << "minimum long = " << LONG_MIN << endl;
  cout << "maximum long = " << LONG_MAX << endl;
  cout << "minimum unsigned long = 0" << endl;
  cout << "maximum unsigned long = " << ULONG_MAX << endl;
}

```

```

minimum short = -32768
maximum short = 32767
maximum unsigned short = 0
maximum unsigned short = 65535
minimum int = -2147483648
maximum int = 2147483647
minimum unsigned int = 0
maximum unsigned int = 4294967295
minimum long = -2147483648
maximum long = 2147483647
minimum unsigned long = 0
maximum unsigned long = 4294967295

```

头文件 <climits> 中定义了 SHRT\_MIN, SHRT\_MAX, USHRT\_MIN 等常量。这些是对应变量的取值范围。如, 输出显示在这台计算机上 int 类型变量的取值范围是: -2 147 483 648 到 2 147 483 647。

在这台计算机上, 3 个 signed 整型与它们所对应的没有限定字的整型范围相同。如

signed short int 与 short int 相同，这表明 signed 整型在这台计算机上是多余的。

输出还显示 int 型（-2 147 483 648 至 2 147 483 647）与 long int 型的范围相同。unsigned int 型（0~4 294 967 295）与 unsigned long int 型范围相同。这表明 long 型在这台机器上是多余的。

例 2.3 的输出表明，在这台计算机（运行 Windows98 及 CodeWarrior 3.2 C++ 编译器的 Pentium II PC 机）上 6 种整型的范围如下。

<b>short:</b>	- 32, 768 ~ 32, 767;	( $2^8$ 个值 = > 1 字节)
<b>int:</b>	- 2, 147, 483, 648 ~ 2, 147, 483, 647;	( $2^{32}$ 个值 = > 4 字节)
<b>long:</b>	- 2, 147, 483, 648 ~ 2, 147, 483, 647;	( $2^{32}$ 个值 = > 4 字节)
<b>unsigned short:</b>	0 ~ 65, 535;	( $2^8$ 个值 = > 1 字节)
<b>unsigned int:</b>	0 ~ 4, 294, 967, 295;	( $2^{32}$ 个值 = > 4 字节)
<b>unsigned long:</b>	0 ~ 4, 294, 967, 295;	( $2^{32}$ 个值 = > 4 字节)

注意 long 与 int 相同，unsigned long 与 unsigned int 相同。

unsigned 整型用于位串。位串就是在计算机的随机存储器（RAM）或磁盘上存储的由 0 和 1 组成的串。当然，所有存储在计算机中（无论是 RAM 还是磁盘中）的数据都是由 0 和 1 组成的。但其他类型的数据都是有格式的，也就是说可以被解释为诸如符号整数或字符串之类。

## 2.6 数学运算符

发明计算机是用做数值计算的。和大多数其他程序设计语言一样，C++ 通过 5 个数学运算符完成数值计算：+、-、\*、/和%。

### 例 2.4 整数运算

这个例子说明数学运算符是如何工作的。

```
int main ()
| // 测试操作符 +, -, *, /, and %:
  int m = 54;
  int n = 20;
  cout << "m = " << m << " and n = " << n << endl;
  cout << "m+n = " << m+n << endl;    // 54+20 = 74
  cout << "m-n = " << m-n << endl;    // 54-20 = 34
  cout << "m*n = " << m*n << endl;    // 54*20 = 1080
  cout << "m/n = " << m/n << endl;    // 54/20 = 2
  cout << "m%n = " << m%n << endl;    // 54%20 = 14
```

```
m = 54 and n = 20
m+n = 74
m-n = 34
m*n = 1080
m/n = 2
m%n = 14
```

注意，整数除的结果还是整数： $54/20 = 2$  而不是 2.7。

## 2.7 增量运算符和减量运算符

整型对象的值可以分别用 `++` 和 `--` 进行增加和减少。每个运算符都有两类：“前”与“后”。“前”类在对象被使用前进行操作（加 1 或减 1），“后”类在对象被使用后进行操作。

### 例 2.5 应用前增量和后增量运算符

```
int main ()
{ // 显示 m++ 和 ++m 的区别:
  int m, n;
  m = 44;
  n = ++m; // 提前加操作符应用于 m
  cout << "m = " << m << ", n = " << n << endl;
  m = 44;
  n = m++; // 提前加操作符应用于 m
  cout << "m = " << m << ", n = " << n << endl;
}
m = 45, n = 45
m = 45, n = 44
```

语句：

```
n = ++m; // 提前加操作符应用于 m
```

先将 `m` 增加为 45，再赋给 `n`。所以当下一条语句打印时，两个变量的值都是 45。

语句：

```
n = m++; // 提前加操作符应用于 m
```

在将 `m` 的值赋给 `n` 后，才将 `m` 增加为 45，所以当下一条语句打印时，`n` 的值为 44。

## 2.8 组合赋值运算符

C++ 的标准赋值运算符是等号 `=`，除了这个运算符外，C++ 还包含下面的组合赋值运算符：`+=`，`-=`，`*=`，`/=`，`%=`。当应用于左边变量的时候，它们对左边变量与右边的表达式运用该数学运算符。

### 例 2.6 使用组合赋值运算符

```
int main ()
{ // 测试算式赋值操作符
  int n = 22;
  cout << "n = " << n << endl;
  n += 9; // n 加 9
```



```

cout << "After n += 9, n = " << n << endl;
n -= 5;    // n减5
cout << "After n -= 5, n = " << n << endl;
n *= 2;    // n乘以2
cout << "After n *= 2, n = " << n << endl;
n /= 3;    // n除以3
cout << "After n /= 3, n = " << n << endl;
n %= 7;    // n除以7以后, 保留的余数
cout << "After n %= 7, n = " << n << endl;

```

```

n = 22
After n += 9, n = 31
After n -= 5, n = 26
After n *= 2, n = 52
After n /= 3, n = 17
After n %= 7, n = 3

```

## 2.9 浮点型

C++ 支持 3 种实数型: float、double 和 long double。在大多数系统中, double 使用 float 两倍的字节数。典型的是 float 使用 4 字节, double 使用 8 字节, long double 使用 8、10、12 或 16 字节。

表示实数的类型由于它们在计算机里存储的方式而被称为“浮点”型。在多数系统中, 像 123.45 这样的数会首选被转换成二进制形式:

$$123.45 = 1111011.01110011_2 \times 2^7$$

然后小数点被“浮动”到使所有的位都在它右边的地方。在这个例子中, 小数点向左移动 7 位形成浮点数格式, 它的尾数是原数的  $1/2^7$ 。所以原数变为:

$$123.45 = 0.111101101110011_2 \times 2^7$$

这个数在内部的表现是: 将尾数 111101101110011 和指数 7 分开进行存储。对于 32 位的 float 型, 23 位用于存储尾数, 8 位用于存储指数, 剩下一位用于存放符号。对于 64 位的 double 型, 52 位用于存储尾数, 11 位用于存储指数。

### 例 2.7 浮点运算

本例与例 2.4 基本相同, 重要的差别是这里的变量被定义为 double 型, 而不是 int 型整数。

```

int main ()
| // 测试浮点型运算符 +, -, *, and /:
  double x = 54.0;
  double y = 20.0;
  cout << "x = " << x << " and y = " << y << endl;
  cout << "x+y = " << x+y << endl;    // 54.0 + 20.0 = 74.0
  cout << "x-y = " << x-y << endl;    // 54.0 - 20.0 = 34.0

```

```

    cout << "x*y = " << x*y << endl;    // 54.0 * 20.0 = 1080.0
    cout << "x/y = " << x/y << endl;    // 54.0/20.0 = 2.7
}
x = 55 and y = 20
x+y = 75
x-y = 35
x*y = 1100
x/y = 2.7

```

和整数除不同，浮点数相除不会截断结果，如： $54.0/20.0 = 2.7$ 。

下面的例子可以用来确定在某台计算机中每种类型所使用的字节数。这个程序使用了 `sizeof` 运算符，它返回指定类型的字节数。

### 例 2.8 使用 `sizeof` 运算符

```

int main ()
{ // 打印基础类型的存储大小
    cout << "Number of bytes used: \n";
    cout << "\t char: " << sizeof (char) << endl;
    cout << "\t short: " << sizeof (short) << endl;
    cout << "\t int: " << sizeof (int) << endl;
    cout << "\t long: " << sizeof (long) << endl;
    cout << "\t unsigned char: " << sizeof (unsigned char) << endl;
    cout << "\t unsigned short: " << sizeof (unsigned short) << endl;
    cout << "\t unsigned int: " << sizeof (unsigned int) << endl;
    cout << "\t unsigned long: " << sizeof (unsigned long) << endl;
    cout << "\t signed char: " << sizeof (signed char) << endl;
    cout << "\t float: " << sizeof (float) << endl;
    cout << "\t double: " << sizeof (double) << endl;
    cout << "\t long double: " << sizeof (long double) << endl;
}

```

```

Number of bytes used:
    char: 1
    short: 2
    int: 4
    long: 4
    unsigned char: 1
    unsigned short: 2
    unsigned int: 4
    unsigned long: 4
    signed char: 1
    float: 4
    double: 8
    long double: 8

```

上面的输出显示了一个典型的 UNIX 工作站上的字节数。这台机器上，`int` 和 `long` 相等，`unsigned int` 与 `unsigned long` 相等，`double` 和 `long double` 相等。换句话说，“长”型与“普通”型之间没有什么区别。

下面的例子可以被用来研究某台计算机中的浮点型，它从 `< cfloat >` 头文件中读取多个

常量值。为了能访问这个头文件，程序中必须包含如下预处理伪指令：

```
#include <cfloat>
```

这就像我们为了使用 cin 与 cout 对象，则必须包含 “#include <iostream>” 伪指令一样。

### 例 2.9 从 `<cfloat>` 头文件中读

```
#include <cstdio> //定义 FLT 常量
#include <iostream> //定义 FLT 常量
using namespace std;
int main ()
{ // 打印基础类型的存储
    int fbits = 8*sizeof (float); // each byte contains 8 bits
    cout << "float uses " << fbits << " bits: \n\t"
        << FLT_MANT_DIG - 1 << " bits for its mantissa, \n\t"
        << fbits - FLT_MANT_DIG << " bits for its exponent, \n\t"
        << 1 << " bit for its sign\n"
        << "          to obtain: " << FLT_DIG << " sig. digits\n"
        << " with minimum value: " << FLT_MIN << endl
        << " and maximum value: " << FLT_MAX << endl;
```

```
float uses 32 bits:
    23 bits for its mantissa,
    8 bits for its exponent,
    1 bit for its sign
    to obtain: 6 sig. digits
with minimum value: 1.17549e-38
    and maximum value: 3.40282e+38
```

常量 FLT\_MANT\_DIG、FLT\_DIG、FLT\_MIN 和 FLT\_MAX 是在头文件 `<float.h>` 中定义的。

这里的输出来自一台 UNIX 工作台，它显示存储 float 所使用的 32 位被分为 3 个部分：23 位用于存放尾数；8 位用于存放指数；1 位用于存放符号。23 位的尾数产生了 6 个有效数字，8 位的指数产生从  $10^{-37}$  至  $3 \times 10^{38}$  的巨大范围，也就是对于任何定义为 float 的变量 x，有：

$$0.000\ 000\ 000\ 0\ 00\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 1 < |x| < 300\ 000\ 000\ 000\ 000\ 000$$
  

$$000\ 000\ 000\ 000\ 000\ 000\ 000$$

所有 float 型的计算都以 double 型的精度进行。所以除非要存储大量的实数而必须考虑存储空间或访问时间, 否则都应该使用 double 型代替 float 型。

## 2.10 类型转换

在第 1 章中已介绍了整数型间是如何自动被转换的。如果需要，C++ 中的整数型也可以被转换为浮点型，如：

```
int n = 22;
```

```
float x = 3.14159;
x += n;           // 值 22 自动转换为 22.0
cout << x - 2 << endl; // 2 自动转换为 2.0
```

这种从整型数向浮点数的转换符合人们的希望。但是从浮点数向整型数的转换却不是自动完成的。

一般情况下，如果  $T$  是一种类型而  $v$  是另一种类型，如表达式：

$T(v)$

可以将  $v$  转换为类型  $T$ ，这称为“类型转换”。如果  $expr$  是一个浮点表达式， $n$  是 `int` 型的变量，则：

```
n = int(expr);
```

会将  $expr$  的值转换成 `int` 型并赋给  $n$ 。这个结果是将实数的小数部分除去，只将剩下的整数部分赋给  $n$ 。如 2.71828 会被转换成 2。注意，这是截断，而不是舍入。

### 例 2.10 简单类型转换

```
int main ()
{ // 把一个双精度型值投影为一个整数
  double v = 1234.56789;
  int n = int(v);
  cout << "v = " << v << ", n = " << n << endl;
}
```

```
// v = 1234.56789; n = 1234
```

当一种类型被转换为更“高”类型时，无需使用转换运算符。这被称为类型提升。这里有一个简单的例子，将 `char` 型一直提升到 `double` 型。

### 例 2.11 类型提升

本例将 `char` 型提升到 `short` 型，再到 `int` 型，再到 `float` 型，再到 `double` 型：

```
int main ()
{ // 打印 65 的从字符到双精度 m 的改进值
  char c = 'A';   cout << " char c = " << c << endl;
  short k = c;    cout << " short k = " << k << endl;
  int m = k;      cout << " int m = " << m << endl;
  long n = m;     cout << " long n = " << n << endl;
  float x = m;    cout << " float x = " << x << endl;
  double y = x;   cout << "double y = " << y << endl;
}
```

```
char c = A
short k = 65
int m = 65
long n = 65
float x = 65
double y = 65
```

字符 A 的整型值是它的 ASCII 码 65。这个值被转换为 char 型的 c, short 型的 k, int 型的 m, long 型的 n。然后这个值被转换成浮点值 65.0 并被以 float 型存放在 x, 以 double 存放在 y 中。注意, cout 将整数 c 打印为字符, 实数 x 与 y 被打印得像整数是因为它们的小数部分都是 0。

由于在 C++ 中很容易在整型数与实数之间进行转换, 因此, 很容易忘掉它们之间的区别。一般来说, 整数用于对离散的事物进行计数, 而实数用于对连续尺度进行度量。这意味着整数值是确切的, 而实数值是近似的。

注意, 类型的转换与提升可以将变量或表达式的值进行转换, 但不能改变变量本身的类型。

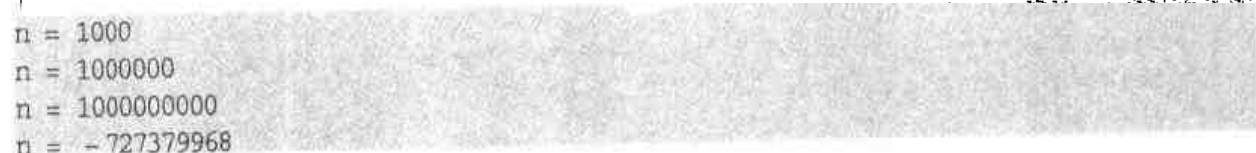
在 C 语言中, 将 v 转换成类型 T 的语法是 (T) v。C++ 继承了这种形式, 所以可以将 `n = int(v)` 写成 `n = (int) v`。

## 2.11 数值溢出

在多数计算机上, long int 型允许 4 294 967 296 个不同的值。这是一个很大的值, 但仍然是有限的。计算机本身就是有限的, 所以, 任何类型的范围都是有限的。但是在数学中, 有很多整数是无限的。因此, 当数值变得过大时, 计算机显然会出错。这种错误被称为数值溢出。

### 例 2.12 整型溢出

```
int main ()
{ // 打印 n 直到它溢出
    int n = 1000;
    cout << "n = " << n << endl;
    n *= 1000; // n 乘以 1000
    cout << "n = " << n << endl;
    n *= 1000; // n 乘以 1000
    cout << "n = " << n << endl;
    n *= 1000; // n 乘以 1000
    cout << "n = " << n << endl;
}
```



```
n = 1000
n = 1000000
n = 1000000000
n = -727379968
```

这表明运行这个程序的计算机不能正确处理 1,000,000,000 与 1000 相乘。

### 例 2.13 浮点型溢出

```
int main ()
{ // 打印 x 直到它溢出
    float x = 1000.0;
```

```

cout << "x = " << x << endl;
x * = x;    // n 乘以 n, 也就是  $x^2$ 
cout << "x = " << x << endl;
x * = x;    // n 乘以 n, 也就是  $x^2$ 
cout << "x = " << x << endl;
x * = x;    // n 乘以 n, 也就是  $x^2$ 
cout << "x = " << x << endl;
x * = x;    // n 乘以 n, 也就是  $x^2$ 
cout << "x = " << x << endl;
}
x = 1000
x = 1e+06
x = 1e+12
x = 1e+24
x = inf

```

这显示从  $x = 1000$  开始, 这台计算机不能第三次以上正确处理  $x$  的平方。最后的输出是代表“无穷大”的特殊符号 `inf`。

注意, 整型数溢出与浮点数溢出之间的差别。例 2.12 中最后的输出是负值  $-727\,379\,968$  而不是正确的值  $1\,000\,000\,000\,000 = 10^{12}$ 。例 2.13 中最后的输出是无穷大的符号 `inf` 而不是正确的值  $10^{48}$ 。整型数的溢出会“反绕”成负数。浮点型数的溢出“汇集”成抽象概念无穷大。

## 2.12 舍入错

舍入错是另一种计算机在对有理数进行数学计算时会发生的错误。如, 数字  $1/3$  会被存储为  $0.333333$ , 但这个数实际上并不等于  $1/3$ 。这种差别被称为舍入错。在某些情况下, 这种错误可能会引起严重的问题。

### 例 2.14 舍入错

本例用一些简单的数学计算说明舍入错:

```

int main ()
| // 显示舍入错
  double x = 1000/3.0; cout << "x = " << x << endl;    // x = 1000/3
  double y = x - 333.0; cout << "y = " << y << endl;    // y = 1/3
  double z = 3 * y - 1.0; cout << "z = " << z << endl;    // z = 3 (1/3) - 1
  if (z == 0) cout << "z = 0. \n";
  else cout << "z does not equal 0. \n";                // z != 0
}
x = 333.333
y = 0.333333
z = -5.68434e-14
z does not equal 0.

```

在严格的数学计算中, 变量应该具有的值是  $x = 333\,1/3$ ,  $y = 1/3$ ,  $z = 0$ 。但是  $1/3$  无法用浮点数准确表示, 这种不准确反映为  $z$  中还有剩余的数。

例 2.14 说明了在用浮点数进行相等条件的判断时所固有的问题。如果  $z$  的值非常接近 0 时, 条件 ( $z == 0$ ) 也会失败 (这种情况可能发生在  $z$  的值在代数中应该为 0 时)。所以最好避免使用浮点数进行相等条件的测试。

下面的例子显示舍入错可能很难被察觉。

### 例 2.15 隐藏的舍入错

本例实现了解二次方程式的二次公式。

```
#include <cmath> // 定义 sqrt () 函数
#include <iostream>
using namespace std;
int main ()
{ // 实现二次方程公式
    float a, b, c;
    cout << "Enter the coefficients of a quadratic equation:" << endl;
    cout << "\ta: ";
    cin >> a;
    cout << "\tb: ";
    cin >> b;
    cout << "\tc: ";
    cin >> c;
    cout << "The equation is:" << a << " * x * x + " << b
        << " * x + " << c << " = 0" << endl;
    float d = b * b - 4 * a * c; // 判别式
    float sqrt d = sqrt (d);
    float x1 = (-b + sqrt d) / (2 * a);
    float x2 = (-b - sqrt d) / (2 * a);
    cout << "The solutions are:" << endl;
    cout << "\tx1 = " << x1 << endl;
    cout << "\tx2 = " << x2 << endl;
    cout << "check:" << endl;
    cout << "\ta * x1 * x1 + b * x1 + c = " << a * x1 * x1 + b * x1 + c << endl;
    cout << "\ta * x2 * x2 + b * x2 + c = " << a * x2 * x2 + b * x2 + c << endl;
}
```

二次公式中需要计算平方根  $\sqrt{b^2 - 4ac}$ , 这是在

```
float sqrt d = sqrt (d);
```

中进行计算的, 它调用了 `<cmath.h>` 头文件中定义的平方根函数 `sqrt ()`。最后两行将解代回原来的公式进行检验, 如果左边的表达式值为 0, 则说明解正确。

方程  $2x^2 + 1x - 3 = 0$  的解是正确的

```
Enter the coefficients of a quadratic equation:
a: 2
b: 1
c: -3
```

```
The equation is : 2*x*x + 1*x + -3 = 0
the solutions are:
```

```
x1 = 1
x2 = -1.5
```

```
Check:
```

```
a*x1*x1 + b*x1 + c = 0
x*x2*x2 + b*x2 + c = 0
```

但是尝试对方程  $x^2 + 10000000000x + 1 = 0$  求解时却失败了:

```
Enter the coefficients of a quadratic equation:
```

```
a: 1
b: 1e10
c: 1
```

```
The equation is : 1*x*x + 1e10*x + 1 = 0
```

```
The solutions are:
```

```
x1 = 0
x2 = -1e10
```

```
Check:
```

```
a*x1*x1 + b*x1 + c = 1
a*x2*x2 + b*x2 + c = 1
```

第一个解  $x_1 = 0$  显然是错误的: 二次公式  $ax_1^2 + bx_1 + c$  的值为 1 而不是 0。第二个解  $x_2 = -1e10 = -10\,000\,000\,000$  甚至更差。正确的解应该是  $x_1 = -0.00000\,00000\,99999\,99999\,99999\,99519$ ,  $x_2 = 9\,999\,999\,999.99999\,99999$ 。

数值溢出与舍入错都属于“运行时错”，这是一种在程序运行时出现的错误。这样的错误比忘记声明变量或漏掉分号之类的编译时错更严重，因为它们通常比较难以发现和定位。编译时错可以被编译器所发现，并给出错误位置很好的说明。但是运行时错只能在用户发现运行结果不正确的时候才能察觉。但就算程序崩溃了，也很难发现哪里出了问题。

## 例 2.16 其他类型的运行时错

这里还有另外两个求例 2.15 中的二次方程式的例子:

```
Enter the coefficients of a quadratic equation:
```

```
a: 1
b: 2
c: 3
```

```
The equation is : 1*x*x + 2*x + 3 = 0
```

```
The solutions are:
```

```
x1 = nan
x2 = nan
```

```
Check:
```

```
a*x1*x1 + b*x1 + c = nan
a*x2*x2 + b*x2 + c = nan
```

二次方程  $1x^2 + 2x + 3 = 0$  没有数解，因为判别式  $b^2 - 4ac$  为负。当程序运行时，求平方根函数  $\text{sqrt}(d)$  由于  $d < 0$  而失败。它返回一个代表“非数字”的常量  $\text{nan}$ 。这之后所有用到这个常量的数学运算都返回同样的值。这就是为什么程序最后检验解的时候会得到



nan。

下例尝试解方程  $0x^2 + 2x + 5 = 0$ ，这个方程有解  $x = 2.5$ 。但是二次公式由于  $a = 0$  而失败。

```
Enter the coefficients of a quadratic equation:
```

```
a: 0
```

```
b: 2
```

```
c: 5
```

```
The equation is : 0 * x * x + 2 * x + 5 = 0
```

```
The solutions are:
```

```
x1 = nan
```

```
x2 = -inf
```

```
Check:
```

```
a * x1 * x1 + b * x1 + c = nan
```

```
a * x2 * x2 + b * x2 + c = nan
```

注意  $x_1$  为 nan 但  $x_2$  为  $-\text{inf}$ 。符号 inf 代表“无穷大”。这正是用非零的数除以 0 时所得到的。二次公式是这样计算  $x_2$  的：

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} = \frac{-(2) - \sqrt{(2)^2 - 4(0)(5)}}{2(0)} = \frac{-2 - 2}{0} = \frac{-4}{0}$$

这就得到  $-\text{inf}$ 。但它是这样计算  $x_1$  的：

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} = \frac{-(2) + \sqrt{(2)^2 - 4(0)(5)}}{2(0)} = \frac{-2 + 2}{0} = \frac{0}{0}$$

这样得到 nan。

inf、 $-\text{inf}$  和 nan 这三个符号是数值常量。通常的数值运算符可以用于它们，但结果通常是没有意义的。如可以将 nan 与一个数相乘，但结果仍然是 nan。

## 2.13 浮点数的电子格式

当输入或输出时，浮点数可以被指定成两种格式：定点记数法和科学记数法。例 2.16 说明了这两种记数法：333.333 是定点记数法而  $-5.68434\text{e}-14$  是科学记数法。

在科学记数法中，字母 e 代表“以 10 为底的指数”。所以  $\text{e}-14$  意味是  $10^{-14}$ ，而  $-5.68434\text{e}-14$  意味是  $-5.68434 \times 10^{-14} = -0.0000000000000568434$ 。显然，用科学记数法可以更有效地记录非常小或非常大的数。

从 0.1 至 999.999 范围的浮点数将按定点记数法正常打印；其他的数都将以科学记数法打印。

### 例 2.17 科学记数法

本例显示如何用科学记数法输入浮点数：

```
int main ()
{ // 用科学型的 e 格式打印双精度值
  double x;
  cout << "Enter float: ";   cin >> x;
  cout << "Its reciprocal is: " << 1/x << endl;
}
```

Enter float: 234.567e89

Its reciprocal is: 4.26317e-92

在科学记数法中可以使用 e 或 E。

## 2.14 作用域

作用域就是程序中一个标识可以起作用的部分。如，变量不能在声明之前被使用，因此，它们的范围是从声明处开始的。下面给出示例。

### 例 2.18 变量的作用域

```
int main ()
{ // 显示变量的范围
  x = 11;    // ERROR: 这不在 x 的范围
  int x;
  { x = 22;  // OK: 这在 x 的范围
    y = 33;  // 这不在 y 的范围
    int y;
    x = 44;  // OK: 这在 x 的范围
    y = 55;  // OK: 这在 y 的范围
  }
  x = 66;    // OK: 这在 x 的范围
  y = 77;    // ERROR: 这不在 y 的范围
}
```

变量 x 的作用域从它被声明处到 main () 函数的结尾处。y 的作用域从它被声明处到内层程序块的结束处。

一个程序可以有多个名字相同的对象，只要它们的域是嵌套的或不相交的。下面给出了一个示例。

### 例 2.19 嵌套和平行的作用域

```
int x = 11;                                // 这个 x 是全局的

int main ()
{ // illustrates the nested and parallel scopes:
  int x = 22;
  { // 内部模块范围的开始
    int x = 33;
    cout << "In block inside main (): x = " << x << endl;
    // 内部模块范围结束
  }
}
```

```

cout << "In main (): x = " << x << endl;
cout << "In main (): :: x = " << :: x << endl;
// 结束 main () 函数的范围
}
In block inside main (): x = 33
In main (): x = 22
In main (): :: x = 11

```

这个程序中有 3 个以  $x$  为名字的不同对象。以值 11 初始化的变量  $x$  是全局变量，所以它的作用域是整个文件。以值 22 初始化的变量  $x$  的作用域被限制在 `main()` 中。由于嵌套在第一个变量  $x$  的作用域范围内，所以它在 `main()` 中将第一个  $x$  隐藏起来了。以值 33 初始化的变量  $x$  作用域是在 `main()` 中的内层程序块，所以在这里第一个和第二个  $x$  都被隐藏起来了。

程序的最后一行使用了作用域运算符访问 `main()` 中隐藏的全局  $x$ 。

## 复 习 题

2.1 写出一条 C++ 语句，在变量 `count` 大于 100 时显示 “Too many”。

2.2 下列代码有什么错误：

```

a. cin << count;
b. if x < y min = x
    else min = y;

```

2.3 这段程序有什么错误：

```

cout << "Enter n: ";
cin >> n;
if (n < 0)
    cout << "That is negative. Try again." << endl;
    cin >> n;
else
    cout << "o.k. n=" << n << endl;

```

2.4 保留字与标准标识符之间有什么区别？

2.5 下列代码有什么错误：

```

enum Semester {FALL, SPRING, SUMMER};
enum Season {SPRING, SUMMER, FALL, WINTER};

```

2.6 下列代码有什么错误：

```

enum Friends {"Jerry", "Henry", "W.D."};

```

## 习 题

2.1 仿造例 2.2 编写并运行一程序，打印元音字母大、小写 10 种形式的 ASCII 码。利用附

录 A 检查程序的输出。

- 2.2 修改 28 页的例 2.15, 使用 double 型代替 float 型 看看程序对于舍入错的输入是不是更好。
- 2.3 编写并运行一个程序, 找出哪种数学运算符可能将变量的值由 3 个数值常量 inf、-inf 和 nan 改变为其他的值。
- 2.4 编写一个将英尺转化成厘米的程序。如, 如果用户要输入 16.9ft, 则结果是输入 42.926cm (1ft 等于 2.54cm)。

## 复习题答案

- 2.1 `if (count > 100) cout << "Too many";`
- 2.2 a. 或者将 `cin` 改为 `cout`, 或者将插入运算符 `<<` 改为提取运算符 `>>`。  
b. 条件 `x < y` 需要圆括号, `if` 分支结束处, `else` 之前应该有分号。
- 2.3 `if` 分支与 `else` 分支都有多条语句, 它们应该被用 `{}` 包括, 组成复合语句。
- 2.4 保留字是程序设计语言保留的关键字, 它们被用于标记语句的结构。如关键字 `if` 和 `else` 是保留字, 标准标识符是定义类型的关键字。在 C++ 的 63 个关键字中, `if`、`else` 和 `while` 是保留字, `char`、`int` 和 `float` 是标准标识符。
- 2.5 第二个 `enum` 定义试图重新定义常量 `SPRING`、`SUMMER` 和 `FALL`。
- 2.6 枚举值必须是有效的标识符。如 "Jerry" 和 "Henry" 的字符串不是标识符。

## 习题答案

```
2.1 int main()
    { // 打印元音的 ASCII 码
      cout << "int('A') = " << int('A') << endl;
      cout << "int('E') = " << int('E') << endl;
      cout << "int('I') = " << int('I') << endl;
      cout << "int('O') = " << int('O') << endl;
      cout << "int('U') = " << int('U') << endl;
      cout << "int('a') = " << int('a') << endl;
      cout << "int('e') = " << int('e') << endl;
      cout << "int('i') = " << int('i') << endl;
      cout << "int('o') = " << int('o') << endl;
      cout << "int('u') = " << int('u') << endl;
    }
```

```
int ('A') = 65
int ('E') = 69
int ('I') = 73
int ('O') = 79
int ('U') = 85
int ('a') = 97
```

```
int ('e') = 101
int ('i') = 105
int ('o') = 111
int ('u') = 117
```

```
2.2 int main()
{ // 实现二次方程式的分式
  double a, b, c;
  cout << "Enter the coefficients:" << endl;
  cout << "\ta: ";
  cin >> a;
  cout << "\tb: ";
  cin >> b;
  cout << "\tc: ";
  cin >> c;
  cout << "The equation is: " << a << "*x*x + " << b
    << "*x + " << c << " = 0" << endl;
  double d = b*b - 4*a*c;
  double sqrt d = sqrt(d);
  double x1 = (-b + sqrt d)/(2*a);
  double x2 = (-b - sqrt d)/(2*a);
  cout << "The solutions are:" << endl;
  cout << "\tx1 = " << x1 << endl;
  cout << "\tx2 = " << x2 << endl;
  cout << "Check:" << endl;
  cout << "\ta*x1*x1 + b*x1 + c = " << a*x1*x1 + b*x1 + c << endl;
  cout << "\ta*x2*x2 + b*x2 + c = " << a*x2*x2 + b*x2 + c << endl;
}
```

```
Enter the coefficients of a quadratic equation:
a: 2
b: 8.001
c: 8.002
The equation is: 2*x*x + 8.001*x + 8.002 = 0
The solutions are:
x1 = -2
x2 = -2.0005
Check:
a*x1*x1 + b*x1 + c = 0
a*x2*x2 + b*x2 + c = 0
```

2.3 下面的程序将变量  $x$  的值从  $\text{inf}$  改变成  $-\text{inf}$ ，反过来也成立。但没有数学运算符可以将变为  $\text{nan}$  的变量值改变。

```
int main()
{ // 在  $x$  变为无穷大以后改变  $x$  的值
  float x=1e30;
  cout << "x= " << x << endl;
  x *= x;
  cout << "x= " << x << endl;
  x *= -1.0;
  cout << "x= " << x << endl;
```

```
x *= -1.0;  
cout << "x= " << x << endl;  
}
```

```
x= 1e+30  
x= inf  
x= -inf  
x= inf
```

#### 2.4 使用 float 型的两个变量。

```
int main()  
{ // 把英文转化为:  
  float inches, cm;  
  cout << "Enter length in inches: ";  
  cin >> inches;  
  cm = 2.54*inches;  
  cout << inches << " inches = " << cm << " centimeters.\n";  
}
```

```
Enter length in inches: 16.9  
16.9 inches = 42.926 centimeters.
```

## 第3章 选 择

前两章中的程序都是顺序执行的，即程序中的每个语句执行一次，并且是按照语句在程序中的排列顺序来执行的。本章介绍如何使用选择语句来编写更加灵活的程序，同时还介绍了 C++ 中使用的各种各样的整数类型。

### 3.1 if 语句

if 语句允许有条件地执行语句。它的语法为：

**if** (条件) 语句；

其中，条件是一个整型表达式；语句是任意的可执行语句。只有当表示条件的整型表达式值为非零时该语句才会被执行。注意，条件必须加圆括号。

#### 例 3.1 可除尽的测试

本例测试一个正整数是否能被另一个正整数除尽：

```
int main ()
{ int n, d;
  cout << "Enter two positive integers: ";
  cin >> n >> d;
  if (n%d) cout << n << " is not divisible by " << d << endl;
}
```

第一次运行时，输入 66 和 7：

```
Enter two positive integers: 66 7
66 is not divisible by 7
```

66%7 的计算结果为 3。由于该值非零，表达式被认为是一个值为 true 的条件，因此，可除尽的信息被打印出来。

第二次运行时，输入 56 和 7：

```
Enter two positive integers: 56 7
```

56%7 的计算结果为 0，被认为是“false”，因此，可除尽的信息不打印。

在 C++ 中，当一个整型表达式被用做条件时，用 0 值表示“false”，而其他任意值都表示“true”。

例 3.1 中的程序并不是很好, 因为其中没有提供当  $n$  可以被  $d$  除尽时的肯定信息, 这个缺点可以使用 `if...else` 语句纠正。

## 3.2 if...else 语句

`if...else` 语句根据条件是否为真在两个语句中选择一个执行。它的语法为:

```
if (条件) 语句 1;
else 语句 2;
```

其中, 条件为一个整型表达式, 语句 1 和语句 2 是任意的可执行语句。如果条件的值为非零, 将执行语句 1; 否则将执行语句 2。

### 例 3.2 可除尽的测试

除了用 `if...else` 语句代替 `if` 语句之外, 本例与例 3.1 中的程序基本是一样的。

```
int main ()
{
    int n, d;
    cout << "Enter two positive integers: ";
    cin >> n >> d;
    if (n%d) cout << n << " is not divisible by " << d << endl;
    else cout << n << " is divisible by " << d << endl;
}
```

现在当输入 56 和 7 时, 可以得到一个确定的回答:

```
Enter two positive integers: 56 7
56 is divisible by 7
```

由于  $56\%7$  的结果为 0, 条件表达式为 `false`, 因此, `else` 后面的语句被执行。注意, 虽然 `if...else` 语句需要两个分号, 它只是一个语句。

## 3.3 关键字

在一种程序设计语言中, 关键字是, 在编程过程中已经定义好的, 并保留作为程序书写的特定用途的单词。标准的 C++ 有 74 个关键字:

<code>and</code>	<code>and_eq</code>	<code>asm</code>	<code>auto</code>	<code>bitand</code>
<code>bitor</code>	<code>bool</code>	<code>break</code>	<code>case</code>	<code>catch</code>
<code>char</code>	<code>class</code>	<code>compl</code>	<code>const</code>	<code>const_cast</code>
<code>continue</code>	<code>default</code>	<code>delete</code>	<code>do</code>	<code>double</code>
<code>dynamic_cast</code>	<code>else</code>	<code>enum</code>	<code>explicit</code>	<code>export</code>
<code>extern</code>	<code>false</code>	<code>float</code>	<code>for</code>	<code>friend</code>
<code>goto</code>	<code>if</code>	<code>inline</code>	<code>int</code>	<code>long</code>
<code>mutable</code>	<code>namespace</code>	<code>new</code>	<code>not</code>	<code>not_eq</code>



operator	or	or_eq	private	protected
public	register	reinterpret_cast	return	short
signed	sizeof	static	static_cast	struct
switch	template	this	throw	true
try	typedef	typeid	typename	using
union	unsigned	virtual	void	volatile
wchar_t	while	xor	xor_eq	

几乎在每一种程序设计语言中都有 if 和 else 等关键字，其他的关键字如 dynamic\_cast 等是 C++ 中仅有的。C++ 的 74 个关键字包括了 C 语言中全部的 32 个关键字。

有两种关键字：保留字和标准标识符。保留字是用做结构标识的关键字，用来定义该语言的语法，如关键字 if 和 else 就是保留字。标准标识符是用来给该语言的特定元素命名的关键字，如关键字 bool 和 int 就是标准标识符，因为它们是 C++ 中的标准数据类型的名称。

有关 C++ 关键字的更多信息参见附录 B。

### 3.4 比较运算符

C++ 中共有六个比较运算符，分别为：

```
x < y      // x 小于 y
x > y      // x 大于 y
x <= y     // x 小于等于 y
x >= y     // x 大于等于 y
x == y     // x 等于 y
x != y     // x 不等于 y
```

这些运算符可以用来比较任意类型表达式的值，运算结果为一个整型表达式，表达式的值是否为 0 可以用做值为真或假的条件。例如，表达式  $7 * 8 < 6 * 9$  的值为 0，表示条件为假。

#### 例 3.3 求两个整数的最小值

本例打印出输入的两个整数中最小的一个：

```
int main ()
| // 打印两个输入值中最小的一个：
  int m, n;
  cout << "Enter two integers: ";
  cin >> m >> n;
  if (m < n) cout << m << " is the minimum." << endl;
  else cout << n << " is the minimum." << endl;
|
```

Enter two integers: 77 55  
55 is the minimum.

注意，在 C++ 中，单个的等号 “=” 是赋值运算符，而两个等号 “==” 是相等运算符：

```
x = 33;      // 将 33 赋给 x
x == 33;     // 值等于 0 (即假)，除非 x 的值为 33。
```

这个区别是非常重要的。

### 例 3.4 一种常见的编程错误

本例是错误的：

```
int main ()
{ // 确定输入值是否等于 22:
  int n;
  cout << "Enter an integer: ";
  cin >> n;
  if (n = 22) cout << n << " = 22" << endl; // 逻辑错误
  else cout << n << " != 22" << endl;
}
```

Enter an integer : 77  
22 = 22

表达式 `n = 22` 将 22 赋值给 `n`，改变了 `n` 原来的值 77。但是表达式 `n = 22` 本身是一个整型表达式，执行后的值为 22。因为只有 0 才表示假，因此，条件 `(n = 22)` 被解释为真，所以，`else` 之前的语句被执行。这一行应该写成：

```
if (n == 22) cout << n << " = 22" << endl; // 正确
```

例 3.4 中的错误被称为逻辑错误，这是一种最糟糕的错误。编译错误（例如，丢了一个分号）是由编译器发现的，运行时错误（例如，以 0 做除数）是由操作系统发现的，但是没有什么手段可以帮助发现逻辑错误。

### 例 3.5 求三个整数的最小值

除了应用于三个整数，本例与例 3.3 基本相同：

```
int main ()
{ // 找出三个输入整数的最小值:
  int n1, n2, n3;
  cout << "Enter three integers: ";
  cin >> n1 >> n2 >> n3;
  int min = n1; // 现在 min = n1
  if (n2 < min) min = n2; // 现在 min = n1 并且 min = n2
  if (n3 < min) min = n3; // 现在 min = n1, min = n2, 并且 min = n3
  cout << "Their minimum is " << min << endl;
}
```

Enter two integers: 77 33 55  
Their minimum is 33

其中的三个注释跟踪了程序的执行过程：`min` 的初值被设为 `n1` 的值，因此，它是数列 `{n1}` 中的最小值。第一个 `if` 语句执行后，`min` 等于 `n1` 和 `n2` 中较小的一个，因此，它是数列 `{n1, n2}` 中的最小值。只有当 `n3` 比当前的 `min` 更小时，最后一个 `if` 语句才将 `min` 的值改为 `n3` 的值。因此，无论何种情况，`min` 总是数列 `{n1, n2, n3}` 中的最小值。

## 3.5 语句块

语句块是用大括号 `{}` 扩起来的语句序列，如下所示：

```
{ int temp = x; x = y; y = temp; }
```

在 C++ 程序中，凡是可以使用单个语句的地方都可以使用语句块。

### 例 3.6 在 if 语句中的语句块

本例输入两个整数，然后以升序输出这两个整数：

```
int main ()
{ // 升序输出两个输入整数
    int x, y;
    cout << "Enter two integers: ";
    cin >> x >> y;
    if (x > y) { int temp=x; x = y; y = temp; } // 交换 x, y
    cout << x << " <= " << y << endl;
}
```

```
Enter two integers: 66 44
```

```
44 <= 66
```

语句块中的三个语句所做的工作为：如果  $x$  和  $y$  的排列顺序不是升序，则交换  $x$  和  $y$ ，使它们按升序排列。这样的交换使用了一个名为 `temp` 的临时存储单元，需要三步完成。通过将这三个语句组成一个语句块，使它们都执行，或者都不执行。

注意，变量 `temp` 是在语句块内部声明的，这使得它成为语句块中的局部变量，也就是说，它仅在该语句块执行过程中存在。如果条件为假（即  $x < y$ ），则 `temp` 将不会存在。这个例子说明了局部对象仅在需要时才被定义。

注意，一个 C++ 程序本身就是一个在前面加上 `int main ()` 的一个语句块。

在 1.5 节中，介绍过一个变量的作用域是能够使用该变量的那部分程序，从变量定义处开始到被该定义控制的块结束，由此，一个语句块可以用来限制变量的作用域，因此，就允许在一个程序的不同部分使用同名的不同变量。

### 例 3.7 使用语句块限制变量的作用域

本例中，三个不同的变量使用了同一个名字 `n`：

```
int main ()
{ // 使用同一名字 n 表示三个不同变量
    int n=44;
    cout << "n = " << n << endl;
    { int n; //范围延长至 4 行
        cout << "Enter an integer: ";
        cin >> n;
    }
```

```

    cout << "n = " << n << endl;
}
cout << "n = " << n << endl;    //开始声明的 n
}
int n;
cout << "n = " << n << endl;    //范围延长至 2 行
}
cout << "n = " << n << endl;    //开始声明的 n
}

```

```

n = 44
Enter an integer: 77
n = 77
n = 44
n = 4251897
n = 44

```

在本例中，有三个内部的语句块。其中，第一个语句块声明了一个新变量  $n$ ， $n$  仅在该语句块中存在，并且覆盖了前面的变量  $n$ ，因此，在新变量  $n$  被输入赋值为 77 时，最初声明的变量  $n$  保持了它的原值 44。第二个语句块没有声明新的变量  $n$ ，所以，最初定义的变量  $n$  的作用域包括该语句块，由此第三个输出的是最初声明的变量  $n$  的值 44。像第一个语句块一样，第三个语句块声明了一个新的变量  $n$  覆盖了最初声明的变量  $n$ 。但是，在第三个语句块中没有初始化它所声明的变量  $n$ ，因此，第四个输出是一个无效值 (4251897)。最后，由于每个变量  $n$  的作用域仅在声明它的语句块中，程序最后一行是在最初声明的变量  $n$  的作用域中，因此打印出 44。

### 3.6 复合条件

可以使用逻辑运算符  $\&\&$  (与)、 $\|\|$  (或) 和  $!$  (非)，把如  $n \% d$  和  $x \geq y$  一样的条件表达式组合为复合的条件。逻辑运算符的定义为：

$p \&\& q$       仅当  $p$  和  $q$  的值都为真时为真  
 $p \|\| q$       仅当  $p$  和  $q$  的值都为假时为假  
 $! p$           仅当  $p$  的值为假时为真

例如，仅当  $n \% d$  值为 0 并且  $x$  小于  $y$  时， $(n \% d \|\| x \geq y)$  的值为假。逻辑运算符的定义通常是由图 3.1 所示的真值表给出的：

p	q	$P \&\& q$
T	T	T
T	F	F
F	T	F
F	F	F

p	q	$P \ \  q$
T	T	T
T	F	T
F	T	T
F	F	F

p	$! p$
T	F
F	T

图 3.1 真值表

这些真值表的意义是：如果  $p$  为真并且  $q$  为假，则表达式  $p \ \&\& \ q$  的值为假，而表达式  $p \ || \ q$  的值为真，依此类推。

下面一个例子的运行结果与例 3.5 相同，不同之处在于它使用了复合条件。

### 例 3.8 使用复合条件

本例与例 3.5 的结果相同，其中使用复合条件来求三个整数的最小值：

```
int main ()
| // 求二个输入整数的最小值
  int n1, n2, n3;
  cout << "Enter three integers: ";
  cin >> n1 >> n2 >> n3;
  if (n1 <= n2 && n1 <= n3) cout << "Their minimum is " << n1 << endl;
  if (n2 <= n1 && n2 <= n3) cout << "Their minimum is " << n2 << endl;
  if (n3 <= n1 && n3 <= n2) cout << "their minimum is " << n3 << endl;
```

Enter two integers: 77 33 55

Their minimum is 33

注意，例 3.8 比起例 3.5 来并没有什么改进，该例的目的只是说明复合条件的使用方法。下面是另一个使用复合条件的例子。

### 例 3.9 友好的用户输入

本例允许用户输入一个“Y”或“y”来表示“yes”：

```
int main ()
| // 以不同形式输入
  char ans;
  cout << "Are you enrolled (y/n): ";
  cin >> ans;
  if (ans == 'Y' || ans == 'y') cout << "You are enrolled. \n";
  else cout << "You are not enrolled. \n";
```

Are you enrolled (y/n): y

You are not enrolled.

在本例中，给用户一个输入答案的提示，并且建议输入“y”或“n”。但是除非用户输入“Y”或“y”，对于用户输入的其他任意字符它都会认为是“no”。

## 3.7 短路

除非需要，使用  $\&\&$  和  $||$  构成的复合条件有时甚至不计算第二个操作数，这称为短路。正如真值表所示，如果  $p$  为假，则条件表达式  $p \ \&\& \ q$  将为假。在这种情况下，就没有必要计算  $q$  了。同样，如果  $p$  为真，则不必计算  $q$  就可以肯定  $p \ || \ q$  为真。上面两种情况下，只要计算出第一个操作数的值，就可以知道整个条件表达式的值。

### 例 3.10 短路

本例测试整除性：

```
int main ()
{ // 测试一个整数是否整除另一个整数
  int n, d;
  cout << "Enter two positive integers: ";
  cin >> n >> d;
  if (d != 0 && n % d == 0) cout << d << " divides " << n << endl;
  else cout << d << " does not divide " << n << endl;
}
```

本次运行，d 为正数并且  $n \% d$  值为 0，所以复合条件为真：

```
Enter two positive integers: 300 6
6 divides 300
```

本次运行，d 为正数但是  $n \% d$  值为非 0，所以复合条件为假：

```
Enter two positive integers: 300 7
6 does not divide 300
```

本次运行，d 为 0，因此没有计算第二个操作数 “ $n \% d == 0$ ”，复合条件立刻被确定为假：

```
Enter two positive integers: 300 0
0 does not divide 300
```

短路可以预防程序崩溃，因为当 d 为 0 时，就不计算  $n \% d$  了。

## 3.8 布尔表达式

布尔表达式是一个值为真或假的条件表达式。在例 3.10 中，表达式  $d > 0$ ， $n \% d == 0$  和  $(d > 0 \&\& n \% d == 0)$  都是布尔表达式。前面曾经介绍过，布尔表达式的值为整数，当值为 0 时表示 “false”，而任一非 0 的值表示 “true”。

由于所有的非 0 整数都被认为是表示 “true”，布尔表达式用起来很灵活。例如：语句

```
if (n) cout << "n is not zero";
```

因为当 n 为非 0 值时，布尔表达式 (n) 的值为真，所以程序恰好打印出 “n is not zero”。这里有一个更实际的例子：

```
if (n % d) cout << "n is not a multiple of d";
```

当  $n \% d$  值为非 0 时，该输出语句可以精确地执行，而当 d 不能整除 n 时，情况是相同的，因为  $n \% d$  是整除后的余数。

布尔表达式值为整数的特点在 C++ 中有时会导致一些异常。

### 例 3.11 另一种逻辑错误

本例是错误的：

```
int main ()
{
    int n1, n2, n3;
    cout << "Enter three integers: ";
    cin >> n1 >> n2 >> n3;
    if (n1 >= n2 >= n3) cout << "max = x"           // 逻辑错误
    ;
    Enter an integer: 0 0 1
    max = 0
```

程序出错的根源就在于布尔表达式的值为整数。表达式  $(n1 \geq n2 \geq n3)$  是从左往右计算的，由于  $0 \geq 0$ ，第一部分  $n1 \geq n2$  的值为“true”。但是“true”被存为数值1。接着该值与同样值为1的  $n3$  相比，因此，即使整个表达式的值为假，表达式的值也将计算为“true”（0不是0，0和1中的最大值）。

这里的问题是出错的语句在语法上是正确的，因此编译器和操作系统都不能发现该错误。与例3.4中的逻辑错误相似，这是另一种逻辑错误。

例3.11中最关键之处在于：布尔表达式具有数值型的值，所以，复合条件的情况可能很复杂。

## 3.9 选择的嵌套

像复合条件一样，选择语句可用于其他任意语句的地方，因此，选择语句可以在另一个选择语句内部使用，这称为嵌套语句。

### 例 3.12 嵌套的选择语句

本例与例3.10的运行结果相同：

```
int main ()
{
    int n, d;
    cout << "Enter two positive integers: ";
    cin >> n >> d;
    if (d != 0)
        if (n%d == 0) cout << d << " divides " << n << endl;
        else cout << d << " does not divide " << n << endl;
    else cout << d << " does not divide " << n << endl;
}
```

在第一个if子句中嵌套了第二个if .. else语句，因此，只有当d为非0时，第二个if .. else语句才会执行。

注意这里“does not divide”句子被使用了两次。其中，第一次是嵌套在第一个 if .. else 语句的 if 子句中，当 d 非 0 并且  $n \% d$  为 0 时执行；第二次是当 d 为 0 时执行。

当对嵌套的 if .. else 语句进行语法分析时，编译器使用如下的规则：

将 else 与上一个距离最近的未匹配的 if 匹配。

使用这条规则，编译器可以很容易地将代码译为如下结构：

```
if (a > 0)    if (b > 0)    ++a;    else if (c > 0)        //不好的代码风格
if (a < 4)    ++b;    else if (b < 4) ++c;    else --a;    //不好的代码风格
else if (c < 4)    --b;    else    --c;    else a = 0;    //不好的代码风格
```

为提高程序的可读性，也可以这样书写：

```
if (a > 0)
if (b > 0) ++a;
else
    if (c > 0)
        if (a < 4)    ++b;
        else
            if (b < 4) ++c;
            else --a;
        else
            if (c < 4) --b;
            else --c;
    else a = 0;
```

或这样书写：

```
if (a > 0)
    if (b > 0) ++a;
    else if (c > 0)
        if (a < 4)    ++b;
        else if (b < 4) ++c;
        else --a;
    else if (c < 4) --b;
    else --c;
else a = 0;
```

在第二种书写方式中，当 else 和 if 形成平行结构时，它们被译做一个整体（参见第 3.10 节）。

### 例 3.13 使用嵌套的选择语句

本例与例 3.5 和例 3.8 的运行结果相同，其中使用了嵌套的 if .. else 语句来求三个整数的最小值：

```
int main ()
{
    int n1, n2, n3;
    cout << "Enter three integers: ";
    cin >> n1 >> n2 >> n3;
```



```

if (n1 < n2)
    if (n1 < n3) cout << "Their minimum is " << n1 << endl;
    else cout << "Their minimum is " << n3 << endl;
else // n1 >= n2
    if (n2 < n3) cout << "Their minimum is " << n2 << endl;
    else cout << "Their minimum is " << n3 << endl;
}
Enter three integers: 77 33 55
Their minimum is 33

```

在本次运行中，第一个条件 ( $n1 < n2$ ) 为假，并且第三个条件 ( $n2 < n3$ ) 为真，因此打印出  $n2$  是最小值。

本例比起例 3.8 来效率较高，因为在任意情况下运行时，它都会仅仅计算两个简单的条件，而不是计算三个复合条件。然而，它是一个较差的程序，因为其逻辑结构比较复杂。在效率和简单性之间，一般情况下最好选择简单性。

### 例 3.14 猜谜游戏

本例可以猜到用户输入的 1 到 8 之间的一个数：

```

int main ()
{
    cout << "Pick a number from 1 to 8." << endl;
    char answer;
    cout << "Is it less than 5? (y|n): "; cin >> ans;
    if (answer == 'y') // 1 <= n <= 4
    { cout << "Is it less than 3? (y|n): "; cin >> answer;
      if (answer == 'y') // 1 <= n <= 2
      { cout << "Is it less than 2? (y|n): "; cin >> answer;
        if (answer == 'y') cout << "Your number is 1." << endl;
        else cout << "Your number is 2." << endl;
      }
    }
    else // 3 <= n <= 4
    { cout << "Is it less than 4? (y|n): "; cin >> answer;
      if (answer == 'y') cout << "Your number is 3." << endl;
      else cout << "Your number is 4." << endl;
    }
    else // 5 <= n <= 8
    { cout << "Is it less than 7? (y|n): "; cin >> answer;
      if (answer == 'y') // 5 <= n <= 6
      { cout << "Is it less than 6? (y|n): "; cin >> answer;
        if (answer == 'y') cout << "Your number is 5." << endl;
        else cout << "Your number is 6." << endl;
      }
    }
    else // 7 <= n <= 8
    { cout << "Is it less than 8? (y|n): "; cin >> answer;
      if (answer == 'y') cout << "Your number is 7." << endl;
      else cout << "Your number is 8." << endl;
    }
}

```

```

    }
}

```

通过不断地将问题细分为子问题，只问三个问题就可以找到 8 个数字中的任意一个。在本次运行中，用户输入的数字是 6。

```

Pick a number from 1 to 8.
Is it less than 5? (y|n): n
Is it less than 7? (y|n): y
Is it less than 6? (y|n): n
Your number is 6.

```

例 3.14 中使用的算法称为“二分法”，该算法可以用更简单的方法实现（参见例 6.14）。

### 3.10 else if 结构

嵌套的 if .. else 语句经常被用来进行一系列平行选项的测试，其中仅 else 子句包含进一步的嵌套。这时，复合的语句通常被排成 else if 短语，以强调逻辑上平行的特征。

#### 例 3.15 使用 else if 结构进行平行选择

本例询问用户的语种，然后用该语种打印一句问候语：

```

int main ()
{ char language;
  cout << "Engl., Fren., Ger., Ital., or Rus.? (e|f|g|i|r): ";
  cin >> language;
  if (language == 'e') cout << "welcome to ProjectEuclid.";
  else if (language == 'f') cout << "Bon jour, ProjectEuclid.";
  else if (language == 'g') cout << "Guten tag, ProjectEuclid.";
  else if (language == 'i') cout << "Bon giorno, ProjectEuclid.";
  else if (language == 'r') cout << "Dobre utre, ProjectEuclid.";
  else cout << "Sorry; we don't speak your language.";
}

```

```

Engl., Fren., Ger., Ital., or Rus.? (e|f|g|i|r): i
Bon giorno, ProjectEuclid.

```

本例使用嵌套的 if .. else 语句来从五个给定的选项中进行选择。

与普通的嵌套 if .. else 语句相同，该程序也可以写成：

```

if (language == 'e') cout << "welcome to ProjectEuclid.";
else
  if (language == 'f') cout << "Bon jour, ProjectEuclid.";
  else
    if (language == 'g') cout << "Guten tag, ProjectEuclid.";
    else
      if (language == 'i') cout << "Bon giorno, ProjectEuclid.";
      else

```

```
if (language == 'r') cout << "Dobre utre, ProjectEuclid.";
else cout << "Sorry; we don't speak your language.";
```

但是前一种格式更好一点，因为它更清楚地表示了逻辑上的平行特征，同时它也使用了较少的锯齿状边缘。

### 例 3.16 使用 else if 结构选择分数的范围

本例将一个测验得分转变为表示级别的等值字母：

```
int main ()
{
    int score;
    cout << "Enter your test score: ";
    cin >> score;
    if (score > 100) cout << "Error: that score is out of range.";
    else if (score >= 90) cout << "Your grade is an A." << endl;
    else if (score >= 80) cout << "Your grade is a B." << endl;
    else if (score >= 70) cout << "Your grade is a C." << endl;
    else if (score >= 60) cout << "Your grade is a D." << endl;
    else if (score >= 0) cout << "Your grade is an F." << endl;
    else cout << "Error: that score is out of range.\n";
}
```

```
Enter your test score: 83
Your grade is a B.
```

程序中使用了一连串的选择语句来测试变量 `score`，直到发现每个条件都为真，或者执行到最后一个 `else` 语句。

## 3.11 switch 语句

`switch` 语句可以用来代替 `else if` 结构来实现一系列的平行选择，它的语法为：

```
switch (表达式)
{
    case 常量 1: 语句序列 1;
    case 常量 2: 语句序列 2;
    case 常量 3: 语句序列 3;
    :
    :
    case 常量 N: 语句序列 N;
    default : 语句序列 0;
}
```

执行过程为：首先计算表达式的值，然后在 `case` 的常量中查找该值。如果在常量列表中找到该值，则执行相应的语句序列中的语句；否则，如果有 `default` 子句（为可选项），则程序分支转向 `default` 子句的语句序列。要注意的是，`switch` 语句中的表达式必须是整型表达式（参见第 2.1 节），并且常量必须为整型常量。

### 例 3.17 使用 switch 语句选择分数的范围

本例与例 3.16 的执行结果相同：

```
int main ()
{ int score;
  cout << "Enter your test score: ";
  cin >> score;
  switch (score/10)
  { case 10:
    case 9: cout << "Your grade is an A." << endl; break;
    case 8: cout << "Your grade is a B." << endl; break;
    case 7: cout << "Your grade is a C." << endl; break;
    case 6: cout << "Your grade is a D." << endl; break;
    case 5:
    case 4:
    case 3:
    case 2:
    case 1:
    case 0: cout << "Your grade is an F." << endl; break;
    default: cout << "Error: score is out of range. \n"
  }
  cout << "Goodbye." << endl;
}
```

```
Enter your test score: 83
Your grade is a B.
Goodbye.
```

在本例中，首先将分数除以 10，使分数范围缩小到 0~10 之间，因此在运行时，分数 83 的值被缩小为 8，程序分支转去执行 case 子句，并且打印出结果。然后，执行 break 语句，使程序分支转到 switch 分支后面的第一条语句处执行，打印出“Goodbye”。

注意，如果分数范围在 101 到 109 或者 -9 到 -1 之间，该程序将产生错误的运行结果。

在 switch 语句中，通常在每个 case 子句的最后加一条 break 语句，否则，程序在执行完一个 case 子句后，将不能直接跳出 switch 语句，而是继续执行该 switch 语句中的下一个 case 子句。通常把这种情况称为一次失败。

### 例 3.18 switch 语句中的失败错误

本例的设计意图与例 3.16 相同，但是由于没有 break 语句，程序执行了所经过的每一个 case 子句：

```
int main ()
{ int score;
  cout << "Enter your test score: ";    cin >> score;
  switch (score/10)
  { case 10:
    case 9: cout << "Your grade is an A." << endl;    //逻辑错误
    case 8: cout << "Your grade is a B." << endl;    //逻辑错误
```

```

    case 7: cout << "Your grade is a C." << endl;    //逻辑错误
    case 6: cout << "Your grade is a D." << endl;    //逻辑错误
    case 5:
    case 4:
    case 3:
    case 2:
    case 1:
    case 0: cout << "Your grade is an F." << endl;    //逻辑错误
    default: cout << "Error: score is out of range. \n";
}
cout << "Goodbye." << endl;
}
Enter your test score: 83
Your grade is a B.
Your grade is a C.
Your grade is a D.
Your grade is an F.
Error: score is out of range.
Goodbye.

```

当程序跳转到 case 8 子句，打印出 “Your grade is a B” 之后，接着执行 case 7 子句，打印 “Your grade is a C”。由于删去了 break 语句，造成失败，程序会一直执行到 default 子句之间的每一个 cout 语句。

## 3.12 条件表达式运算符

C++ 提供了一种特殊的运算符，经常被用来代替 if ... else 语句，称为条件表达式运算符，该运算符的语法中使用了符号 “?” 和 “:”。

条件? 表达式 1: 表达式 2

这是一个三元运算符，由三个操作数产生一个确定的值，根据布尔型的条件值决定其结果是表达式 1 的值或表达式 2 的值。例如，赋值语句

```
min = ( x < y ? x : y );
```

将 x 与 y 的最小值赋给 min，因为如果条件  $x < y$  为真，则表达式  $(x < y ? x : y)$  的值为 x，否则表达式的值为 y。

只有当条件和两个表达式都比较简单时才使用条件表达式语句，所以条件表达式语句用得较少。

### 例 3.19 求最小值的另一个例子

本例与例 3.3 的运行结果相同：

```

int main ()
{ int m, n;
  cout << "Enter two integers: ";

```

```

cin >> m >> n;
cout << (m<n? m: n) << " is the minimum." << endl;
}

```

其中, 如果  $m < n$ , 条件表达式  $(m < n ? m : n)$  的值为  $m$ , 否则为  $n$ 。

## 复 习 题

3.1 写出一个简单的 C++ 语句, 如果变量 `count` 超过 100 则打印 “Too many”。

3.2 以下代码中有什么错误:

```

a. cin << count;
b. if x < y min = x
    else min = y;

```

3.3 以下代码中有什么错误:

```

cout << "Enter n: ";
cin >> n;
if (n < 0)
    cout << "that is negative. Try again." << endl;
    cin >> n;
else
    cout << "o.k. n = " << n << endl;

```

3.4 保留字和标准标识符有什么区别?

3.5 说明以下各题的对错。如果是错误的, 说明为什么。

- a.  $!(p || q)$  与  $!p || !q$  相等
- b.  $!!!p$  与  $!p$  相等
- c.  $p \&\& q || r$  与  $p \&\& (q || r)$  相等

3.6 构造以下各布尔表达式的真值表, 对操作数  $p$  和  $q$  的 4 种真值的组合情况写出表达式的真值 (0 或 1)。

- a.  $!p || q$
- b.  $p \&\& q || !p \&\& !q$
- c.  $(p || q) \&\& ! (p \&\& q)$

3.7 使用真值表判断以下各题中两个布尔表达式的值是否相等。

- a.  $!(p \&\& q)$  和  $!p \&\& !q$
- b.  $!!p$  和  $p$
- c.  $!p || q$  和  $p || !q$
- d.  $p \&\& (q \&\& r)$  和  $(p \&\& q) \&\& r$
- e.  $p || (q \&\& r)$  和  $(p || q) \&\& r$

3.8 什么是短路? 短路有什么作用?

3.9 以下代码有什么错误?

```
if (x == 0) cout << x << "-0\n";  
else cout << x << "! = 0\n";
```

3.10 以下代码有什么错误?

```
if (x < y < z) cout << x << " < " << y << " < " << z << endl;
```

3.11 构造一个逻辑表达式来表示以下的条件:

- a. score 大于等于 80 但小于 90;
- b. answer 为 'N' 或 'n';
- c. n 不等于 8;
- d. ch 是一个大写字母。

3.12 构造一个逻辑表达式来表示以下的条件:

- a. n 在 0 到 7 之间, 但不等于 3;
- b. n 在 0 到 7 之间, 但不是偶数;
- c. n 能被 3 整除, 但不能被 30 整除;
- d. ch 是一个小写字母或大写字母。

3.13 以下代码有什么错误?

```
if (x == 0)  
    if (y == 0) cout << "x and y are both zero." << endl;  
else cout << "x is not zero." << endl;
```

3.14 以下两个语句有什么区别?

```
if (n > 2) {if (n < 6) cout << "OK";} else cout << "NG";  
if (n > 2) {if (n < 6) cout << "OK"; else cout << "NG";}
```

3.15 什么是“fall-through”(失败)?

3.16 以下表达式的值是什么?

```
(x < y ? ~1 : (x == y ? 0 : 1));
```

3.17 写出一个 C++ 语句, 使用条件表达式运算符将 x 的绝对值赋给 absx。

3.18 写出一个 C++ 语句, 如果变量 count 超过 100, 则打印“too many”, 使用:

- a. if 语句;
- b. 条件表达式运算符。

## 习 题

- 3.1 修改例 3.1, 使程序在只有 n 能被 d 整除时才打印出一个提示。
- 3.2 修改例 3.5, 打印出四个输入整数中的最小值。
- 3.3 修改例 3.5, 打印出三个输入整数的中间值。
- 3.4 修改例 3.6, 使程序不使用语句块而运行结果不变。

- 3.5 在例 3.7 中, 推测删除第 5 行中的声明后程序的输出, 然后运行程序来检查你的推测。
- 3.6 写出一个程序并运行, 输入用户的年龄 (age), 如果  $\text{age} < 18$ , 则打印 "You are a child."; 如果  $18 \leq \text{age} < 65$ , 则打印 "You are a adult."; 如果  $\text{age} \geq 65$ , 则打印 "You are a senior citizen."。
- 3.7 写出一个程序并运行, 输入两个整数, 然后使用条件表达式运算符, 根据其中的一个整数是否是另一个整数的倍数打印出 "multiple" 或 "not"。
- 3.8 写出一个模拟简单计算器的程序并运行, 输入两个整数和一个字符, 如果该字符是一个 "+", 则打印和; 如果该字符是一个 "-", 则打印差; 如果该字符是一个 "\*", 则打印积; 如果该字符是一个 "/", 则打印商; 如果该字符是一个 "%", 则打印余数。使用 switch 语句。
- 3.9 写出一个玩 "石头, 剪刀, 布" 游戏的程序并运行。在这个游戏中, 两个人同时说 (或者显示一个手的符号来代表) "石头" 或 "剪刀" 或 "布", 压过另一方的为胜者。规则为: 布压过石头, 石头压过剪刀, 剪刀压过布。其中的选择和结果使用枚举型。
- 3.10 用 switch 语句修改例 3.9。
- 3.11 用条件表达式修改例 3.10。
- 3.12 写出一个解二元方程式的程序并测试。一个二元方程式是形如  $ax^2 + bx + c = 0$  的等式, 其中  $a$ 、 $b$  和  $c$  为给定的系数,  $x$  为未知数。系数是实数, 所以应该声明为 float 或 double 型。由于二元方程式通常有两个解, 使用  $x_1$  和  $x_2$  作为输出的解, 它们可以声明为 double 型以避免由于舍入而造成的错误。
- 3.13 写出一个程序并运行, 输入一个六位的整数然后打印出它的六位数字的和。使用除法运算符/和取余运算符%来抽取出该整数的六位数字, 例如: 如果  $n$  为整数 876 543, 则  $n/1000\%10$  为它的千位数字 6。
- 3.14 修改例 3.17, 使得该程序对所有的输入可以运行出正确的结果。

## 复习题答案

- 3.1 `if (count > 100) cout << "Too many";`
- 3.2 a. 使用 `cout` 代替 `cin`, 或者可以使用提取运算符 `>>` 代替插入运算符 `<<`。  
b. 条件  $x < y$  需要加圆括号, 并且在 `else` 之前的 `if` 子句结尾需要加一个分号。
- 3.3 在 `if` 和 `else` 子句中间有多个语句, 需要使用大括号 `{}` 括起来, 形成一个语句块。
- 3.4 保留字是在一种程序设计语言中用做结构标识的关键字。例如, 关键字 `if` 和 `else` 就是保留字。标准标识符是用来定义一种类型的关键字, 在 C++ 的 63 个关键字中, `if`、`else` 和 `while` 就是保留字, 而 `float` 就是标准标识符。
- 3.5 a.  $!(p \parallel q)$  与  $!p \parallel !q$  不等; 例如, 如果  $p$  为真且  $q$  为假, 则第一个表达式的值为假, 但是第二个表达式的值为真。与  $!(p \parallel q)$  相等的表达式应为  $!p \&\&!q$ 。  
b.  $!!!p$  与  $!q$  相等。



c.  $p \&\& q \parallel r$  与  $p \&\& (q \parallel r)$  不等; 例如, 如果  $p$  为假且  $q$  为真, 则第一个表达式的值为真, 但是第二个表达式的值为假:  $p \&\& q \parallel r$  与  $(p \&\& q) \parallel r$  相等。

3.6 布尔表达式的真值表为:

p	q	$! p \parallel q$	p	q	$p \&\& q \parallel ! p \&\& ! q$	p	q	$(p \parallel q) \&\& ! (p \&\& q)$
T	T	T	T	T	T	T	T	F
T	F	F	T	F	F	T	F	T
F	T	T	F	T	F	F	T	T
F	F	T	F	F	T	F	F	F

3.7 a. 这两个布尔表达式的真值表不相等。

p	q	$! (p \&\& q)$	p	q	$! p \&\& ! q$
T	T	F	T	T	T
T	F	T	T	F	T
F	T	T	F	T	T
F	F	T	F	F	F

b. 这两个布尔表达式的真值表相等。

p	$! pp$	p	p
T	T	T	T
F	F	F	F

c. 这两个布尔表达式的真值表不相等。

p	q	$! p \parallel q$	p	q	$p \parallel ! q$
T	T	T	T	T	T
T	F	F	T	F	T
F	T	T	F	T	F
F	F	T	F	F	T

d. 这两个布尔表达式的真值表相等。

p	q	r	$p \ \&\& \ (q \ \&\& \ r)$	p	q	r	$(p \ \&\& \ q) \ \&\& \ r$
T	T	T	T	T	T	T	T
T	T	F	F	T	T	F	F
T	F	T	F	T	F	T	F
T	F	F	F	T	F	F	F
F	T	T	F	F	T	T	F
F	T	F	F	F	T	F	F
F	F	T	F	F	F	T	F
F	F	F	F	F	F	F	F

e. 这两个布尔表达式的真值表不相等。

p	q	r	$p \    \ (q \ \&\& \ r)$	p	q	r	$(p \    \ q) \ \&\& \ r$
T	T	T	T	T	T	T	T
T	T	F	T	T	T	F	F
T	F	T	T	T	F	T	T
T	F	F	T	T	F	F	F
F	T	T	T	F	T	T	T
F	T	F	F	F	T	F	F
F	F	T	F	F	F	T	F
F	F	F	F	F	F	F	F

3.8 “短路”用于描述在 C++ 中计算像  $(x > 2 \ || \ y > 5)$   $(x > 2 \ \&\& \ y > 5)$  一样的逻辑表达式的方法。在第一个表达式中，如果  $x$  大于 2，则  $y$  将不会被计算。在第二个表达式中，如果  $x$  小于或等于 2，则  $y$  将不会被计算。在这种情况下，只计算复合表达式的第一部分，该部分的值决定了整个复合表达式的真值。

3.9 程序员可能试图测试条件  $(x == 0)$ ，但是由于使用了赋值运算符“=”而不是相等运算符“==”，结果将与设计意图截然不同。例如，在 if 语句之前，如果  $x$  的值为 22，则 if 语句将把  $x$  的值改为 0。此外，赋值表达式  $(x = 0)$  的值将为 0，表示“false”，因此 else 子句将被执行，打印出  $x$  而不是 0。

3.10 程序员可能试图测试条件  $(x < y \ \&\& \ y < z)$ 。程序被编译并运行，但结果与设计意图不同。例如，如果  $x$ 、 $y$  和  $z$  开始的值分别为 44、66 和 22，则代数条件式“ $x < y < z$ ”为假。但是就像程序中所写，该式将从左到右进行计算，即  $(x < y) < z$ 。首先表达式  $(x < y)$  的值为 1，然后复合条件  $(x < y) < z$  将计算  $(1) < 66$ ，其值也是真。因此，输出语句会被执行，错误地输出  $44 < 66 < 22$ 。

3.11 a.  $(score \geq 80 \ \&\& \ score < 90)$

b.  $(answer == 'N' \ || \ answer == 'n')$

c.  $(n \% 2 == 0 \ \&\& \ n != 8)$

d.  $(ch \geq 'A' \ \&\& \ ch \leq 'Z')$

3.12 a.  $(n > 0 \ \&\& \ n < 7 \ \&\& \ n != 3)$

b.  $(n > 0 \ \&\& \ n < 7 \ \&\& \ n \% 2 != 0)$

c.  $((ch \geq 'A' \ \&\& \ ch \leq 'Z') \ || \ (ch \geq 'a' \ \&\& \ ch \leq 'z'))$

3.13 程序员的设计意图很显然是如果第一个条件  $(x == 0)$  为假时, 无论第二个条件  $(y == 0)$  为真或为假, 都输出 “x is not zero.”, 即 else 是要与第一个 if 相匹配的。但是 “else 匹配” 规则使得该 else 与第二个条件匹配, 这意味着只有当 x 为 0 且 y 不为 0 时才输出 “x is not zero.”。可以使用大括号使 “else 匹配” 规则不起作用:

```
if (x == 0)
{ if (y == 0) cout << "x and y are both zero." << endl;
  }
else cout << "x is not zero." << endl;
```

注意, 这里 else 将与第一个 if 匹配, 正如程序员所希望的一样。

3.14 在第一个语句中, else 与第一个 if 匹配。在第二个语句中, else 与第二个 if 匹配。如果  $n \leq 2$ , 第一个语句将打印 NG, 而第二个语句什么都不做。如果  $2 < n < 6$ , 则两个语句都打印 OK。如果  $n \geq 6$ , 则第一个语句将什么都不做, 而第二个语句将打印 NG。注意, 由于没有按照标准的锯齿状习惯书写, 这个程序很难阅读。第一个语句应该写为:

```
if (n > 2)
{ if (n < 6) cout << "OK";
  }
else cout << "NG";
```

这里需要一个大括号来使 “else 匹配” 规则不起作用。else 会与第一个 if 匹配。第二个语句应该写为:

```
if (n > 2)
  if (n < 6) cout << "OK";
  else cout << "NG";
```

这里不需要大括号, 因为设计意图就是使 else 与第二个 if 匹配。

3.15 在 switch 语句中的 “失败” 是指由于没有 break 语句, 因此使得程序继续执行下一个 case 语句。

3.16 如果  $x < y$ , 表达式的值为 -1, 如果  $x == y$ , 表达式的值为 0, 如果  $x > y$ , 表达式的值为 1。

3.17  $absx = (x > 0 ? x : -x);$

3.18 a.  $if (count > 100) cout << "too many";$

b.  $cout << (count > 100 ? "too many" : "");$

## 习题答案

3.1 当  $n$  能被  $d$  整除时, 例 3.1 打印出结果:

```
int main()
{ int n, d;
  cout << "Enter two positive integers: ";
  cin >> n >> d;
  if (n%d == 0) cout << n << " is divisible by " << d << endl;
}
```

Enter two positive integers: 56 7  
56 is divisible by 7

3.2 例 3.5 打印出输入的 4 个整数中的最小值。

```
int main()
{ int n1, n2, n3, n4;
  cout << "Enter four integers: ";
  cin >> n1 >> n2 >> n3 >> n4;
  int min=n1;           // 现在 min<=n1
  if (n2 < min) min = n2; // 现在 min<=n1,n2
  if (n3 < min) min = n3; // 现在 min<=n1,n2,n3
  if (n4 < min) min = n4; // 现在 min<=n1,n2,n3,n4
  cout << "Their minimum is " << min << endl;
}
```

Enter four integers: 44 88 22 66  
Their minimum is 22

3.3 本程序求输入的 3 个整数的中间值。

```
int main()
{ int n1, n2, n3;
  cout << "Enter three integers: ";
  cin >> n1 >> n2 >> n3;
  cout << "Their median is ";
  if (n1 < n2)
    if (n2 < n3) cout << n2;           // n1 < n2 < n3
    else if (n1 < n3) cout << n3;       // n1 < n3 <= n2
    else cout << n1;                     // n3 <= n1 < n2
  else if (n1 < n3) cout << n1;          // n2 <= n1 < n3
  else if (n2 < n3) cout << n2;          // n2 < n3 <= n1
  else cout << n3;                       // n3 <= n2 <= n1
}
```

Enter three integers: 44 88 22  
their median is 44

3.4 本程序与例 3.6 的运行结果相同。

```
int main()
{ int x, y;
  cout << "Enter two integers: ";
```

```

cin >> x >> y;
if (x > y) cout << y << " <= " << x << endl;
else cout << x << " <= " << y << endl;
}

```

Enter two integers: 66 44

44 <= 66

### 3.5 例 3.7 的程序修改为:

```

int main()
{ int n=44;
  cout << "n = " << n << endl;
  { cout << "Enter an integer: ";
    cin >> n;
    cout << "n = " << n << endl;
  }
  { cout << "n = " << n << endl;
  }
  { int n;
    cout << "n = " << n << endl;
  }
  cout << "n = " << n << endl;
}

```

n = 44

Enter an integer: 77

n = 77

n = 77

n = 4251897

n = 77

### 3.6 这里使用了 else if 结构,因为三个输出结果依赖于 age,是三个不相邻的整数之一。

```

int main()
{ int age;
  cout << "Enter your age: ";
  cin >> age;
  if (age < 18) cout << "You are a child.\n";
  else if (age < 65) cout << "You are an adult.\n";
  else cout << "you are a senior citizen.\n";
}

```

Enter your age: 44

You are an adult.

如果控制到达第二个条件 ( $\text{age} < 65$ ), 则第一个条件必须为假, 即  $18 < \text{age} < 65$ 。

同样地, 如果控制到达第二个 else, 则两个条件必须都为假, 即  $\text{age} > = 65$ 。

### 3.7 若整数 m 被整数 n 整除后的余数为 0, 则 m 是 n 的倍数。因此, 复合条件 $m \% n == 0 \mid n \% m == 0$ 测试一个数是否是另一个数的倍数。

```

int main()
{ int m, n;
  cin >> m >> n;
  cout << (m % n == 0 || n % m == 0 ? "multiple" : "not") << endl;
}

```

30.4

not

30.5

multiple

根据复合条件是否为真, 条件表达式的值将为 “multiple” 或 “not”, 因此, 将整个条件表达式传给输出流将产生想要得到的结果。

### 3.8 代表运算符的字符可以用做 switch 语句的控制变量。

```
int main()
{ int x, y;
  char op;
  cout << "Enter two integers: ";
  cin >> x >> y;
  cout << "Enter an operator: ";
  cin >> op;
  switch (op)
  { case '+': cout << x + y << endl; break;
    case '-': cout << x - y << endl; break;
    case '*': cout << x * y << endl; break;
    case '/': cout << x / y << endl; break;
    case '%': cout << x % y << endl; break;
  }
}
```

Enter two integers: 30.13

Enter an operator: %

4

在每个 case 语句中, 简单地打印出相应的数学表达式, 然后中止。

### 3.9 首先定义两个枚举型的 Choice 和 Result, 然后声明枚举型的变量 choice1、choice2 和 result, 同时使用一个整数 n 来取得输入值并将该输入值赋给这些变量:

```
enum Choice {ROCK, PAPER, SCISSORS};
enum Winner {PLAYER1, PLAYER2, TIE};
int main()
{ int n;
  Choice choice1, choice2;
  Winner winner;
  cout << "Choose rock (0), paper (1), or scissors (2):" << endl;
  cout << "Player #1: ";
  cin >> n;
  choice1 = Choice(n);
  cout << "Player #2: ";
  cin >> n;
  choice2 = Choice(n);
  if (choice1 == choice2) winner = TIE;
  else if (choice1 == ROCK)
    if (choice2 == PAPER) winner = PLAYER2;
    else winner = PLAYER1;
  else if (choice1 == PAPER)
    if (choice2 == SCISSORS) winner = PLAYER2;
```

```

    else winner = PLAYER1;
else // (choice1 == SCISSORS)
    if (choice2 == ROCK) winner = PLAYER2;
    else winner = PLAYER1;
if (winner == TIE) cout << "\tYou tied.\n";
else if (winner == PLAYER1) cout << "\tPlayer #1 wins." << endl;
else cout << "\tPlayer #2 wins." << endl;
}

```

Choose rock (0), paper (1), or scissors (2):

Player #1: 1

Player #2: 1

    You tied.

Choose rock (0), paper (1), or scissors (2):

Player #1: 2

Player #2: 1

    Player #1 wins.

Choose rock (0), paper (1), or scissors (2):

Player #1: 2

Player #2: 0

    Player #2 wins.

通过一系列嵌套的 if 语句，可以将所有可能的情况考虑到。

### 3.10 使用 switch 语句:

```

enum Winner {PLAYER1, PLAYER2, TIE};
int main()
{ int choice1, choice2;
  Winner winner;
  cout << "Choose rock (0), paper (1), or scissors (2):" << endl;
  cout << "Player #1: ";
  cin >> choice1;
  cout << "Player #2: ";
  cin >> choice2;
  switch (choice2 - choice1)
  { case 0:
    winner = TIE;
    break;
    case -1:
    case 2:
    winner = PLAYER1;
    break;
    case -2:
    case 1:
    winner = PLAYER2;
  }

  if (winner == TIE) cout << "\tYou tied.\n";
  else if (winner == PLAYER1) cout << "\tPlayer #1 wins." << endl;
  else cout << "\tPlayer #2 wins." << endl;
}

```

### 3.11 使用 switch 语句和条件表达式:

```
enum Winner {PLAYER1, PLAYER2, TIE};
int main()
{ int choice1, choice2;
  cout << "Choose rock (0), paper (1), or scissors (2):" << endl;
  cout << "Player #1: ";
  cin >> choice1;
  cout << "Player #2: ";
  cin >> choice2;
  int n = (choice1 - choice2 + 3) % 3;
  Winner winner = ( n==0 ? TIE : (n==1?PLAYER1:PLAYER2) );
  if (winner == TIE) cout << "\tYou tied.\n";
  else if (winner == PLAYER1) cout << "\tPlayer #1 wins." << endl;
  else cout << "\tPlayer #2 wins." << endl;
}
```

3.12 二元方程式的解由以下公式得出：

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

但当  $a$  为 0 时不能使用该公式，因此条件必须分开检查。如果根号下的表达式值为负时，该公式也不能使用（对实数而言）。表达式  $b^2 - 4ac$  称为公式的判别式，定义一个单独的变量  $d$ ，并且检查它的符号。

```
#include <iostream>
#include <cmath> // 定义 sqrt() 函数
int main()
{ // 解方程 a*x*x+b*x+c=0
  float a, b, c;
  cout << "Enter coefficients of quadratic equation: ";
  cin >> a >> b >> c;
  if (a == 0)
  { cout << "This is not a quadratic equation: a == 0\n";
    return 0;
  }
  cout << "The equation is: " << a << "x^2 + " << b
    << "x + " << c << "= 0\n";
  double d, x1, x2;
  d = b*b - 4*a*c; // 判别式
  if (d < 0)
  { cout << "This equation has no real solutions: d < 0\n";
    return 0;
  }
  x1 = (-b + sqrt(d))/(2*a);
  x2 = (-b - sqrt(d))/(2*a);
  cout << "The solutions are: " << x1 << ", " << x2 << endl;
}
```

```
Enter coefficients of quadratic equation: 2 1 -6
The equation is: 2x^2 + 1x + -6 = 0
The solutions are: 1.5, -2
```



```
Enter coefficients of quadratic equation: 1 4 5
The equation is: 1x2 + 4x + 5 = 0
This equation has no real solutions: d < 0
```

```
Enter coefficients of quadratic equation: 0 4 5
This is not a quadratic equation: a == 0
```

注意，这里如何在选择语句中使用 return 语句来决定是否 a 为 0 或者 d 为负数。这种选择是通过在每个 if 语句中使用一个 else 语句来实现的。

### 3.13 本程序打印出给定的整数各位数字的和：

```
int main()
{ int n, sum;
  cout << "Enter a six-digit integer: ";
  cin >> n;
  sum = n%10 + n/10%10 + n/100%10 + n/1000%10 + n/10000%10
        + n/100000;
  cout << "The sum of the digits of " << n << " is " << sum<<endl;
}
```

```
Enter a six-digit integer: 876543
The sum of the digits of 876543 is 33
```

### 3.14 例 3.7 正确的写法为：

```
int main()
{ // 对一个给定的测试成绩报告用户的级别
  int score;
  cout << "Enter your test score: ";
  cin >> score;
  if (score > 100 || score < 0)
    cout << "Error: that score is out of range.\n";
  else
    switch (score/10)
    { case 10:
      case 9: cout << "Your grade is an A.\n"; break;
      case 8: cout << "Your grade is a B.\n"; break;
      case 7: cout << "Your grade is a C.\n"; break;
      case 6: cout << "Your grade is a D.\n"; break;
      default: cout << "Your grade is an F.\n"; break;
    }
  cout << "Goodbye." << endl;
}
```

```
Enter your test score: 103
Error: that score is out of range.
Goodbye.
```

```
Enter your test score: 93
Your grade is an A.
Goodbye.
```

```
Enter your test score: -3
Error: that score is out of range.
Goodbye.
```

## 第4章 迭 代

迭代是指在程序中重复执行一个语句或语句块。C++中有三个迭代语句：while 语句、do..while 语句和 for 语句。由于它们有循环特征，迭代语句也被称为循环语句。

### 4.1 while 语句

while 语句的语法为：

**while** (条件) 语句；

其中，条件是一个整型表达式，语句是任意的可执行语句。如果条件表达式的值为 0（表示“false”），则不执行该语句，并且程序立即跳转到跟在 while 语句后的下一个语句。如果条件表达式的值为非 0（表示“true”）则重复执行该语句直到条件表达式的值为 0。注意，条件必须用圆括号括起来。

#### 例 4.1 使用 while 循环计算连续整数的和

本例输入一个整数  $n$ ，计算  $1+2+3+\cdots+n$  的和：

```
int main ()
{
    int n, i=1;
    cout << "Enter a positive integer: ";
    cin >> n;
    long sum=0;
    while (i <= n)
        sum += i++;
    cout << "The sum of the first " << n << " integers is "
    << sum;
```

在本例中，使用了三个局部变量： $n$ 、 $i$  和  $sum$ 。每当 while 循环执行一次， $i$  增加 1 然后与  $sum$  相加。当  $i = n$  时循环结束，所以， $n$  是最后一个与  $sum$  相加的值。图 4.1 是跟踪程序执行情况的表格，说明了在用户输入  $n$  的值为 8 后，每次迭代中  $i$  和  $sum$  的值。本次执行的输出为：

i	sum
0	0
1	1
2	3
3	6
4	10
5	15
6	21
7	28
8	36

```
Enter a positive integer: 8
The sum of the first 8 integers is 36
```

图 4.1  $i$  与  $sum$  值的变化

程序计算出  $1 + 2 + 3 + 4 + 5 + 6 - 7 + 8 = 36$ 。

在第二次运行时, 用户输入  $n$  的值为 100, 因此, while 循环迭代 100 次, 计算出  $1 + 2 + 3 + \cdots + 98 + 99 + 100 = 5050$ :

Enter a positive integer: 100  
The sum of the first 100 integers is 5050

注意循环中的语句是设计好的。这个习惯使得程序逻辑很容易跟踪, 特别是在大程序中。

#### 例 4.2 使用 while 循环计算倒数的和

本例计算倒数的和  $s = 1 + 1/2 + 1/3 + \cdots + 1/n$ , 其中  $n$  是大于或等于  $s$  的最小整数:

```
int main ()
{ int bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  double sum = 0.0;
  int i = 0;
  while (sum < bound)
    sum += 1.0 / ++ i;
  cout << "The sum of the first " << i
    << " reciprocals is " << sum << endl;
}
```

当  $n$  的输入值为 3 时, 本次运行计算出  $1 + 1/2 + 1/3 + \cdots + 1/11 = 3.01988$ :

Enter a positive integer: 3  
The sum of the first 11 reciprocals is 3.01988

本次运行的执行情况如图 4.2 所示, 在第 11 次运行之后和才超过 3。

#### 例 4.3 使用 while 循环重复计算

本例打印出用户输入的每个数值的平方根。在程序运行中使用 while 循环进行数值的计算:

```
int main ()
{ double x;
  cout << "Enter a positive number: ";
  cin >> x;
  while (x > 0)
  { cout << "sqrt (" << x << ") = " << sqrt (x) << endl;
    cout << "Enter another positive number (or 0 to quit): ";
    cin >> x;
  }
}
```

i	sum
0	0.00000
1	1.00000
2	1.50000
3	1.83333
4	2.08333
5	2.28333
6	2.45000
7	2.59286
8	2.71786
9	2.82897
10	2.92897
11	3.01988

图 4.2 i 与 sum 的变化

```
Enter a positive integer: 49
sqrt (49) = 7
Enter another positive number (or 0 to quit) : 3.14159
sqrt (3.14159) = 1.77245
Enter another positive number (or 0 to quit) : 100000
sqrt (100000) = 316.228
Enter another positive number (or 0 to quit) : 0
```

在例 4.3 中, 条件 ( $x > 0$ ) 是使用变量  $x$  来控制循环的。它的值通过输入语句在循环内部改变。这样使用的变量被称为循环控制变量。

## 4.2 循环的终止

前面已经介绍了怎样使用 `break` 语句来控制 `switch` 语句 (参见例 3.17), 本节介绍怎样用 `break` 语句来控制循环。

### 例 4.4 使用 `break` 语句终止循环

本例与例 4.1 的运行结果相同:

```
int main ()
{ int n, i = 1;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum = 0;
  while (true)
  { if (i > n) break;    //立即终止循环
    sum += i++;
  }
  cout << "The sum of the first " << n << " integers is " << sum;
```

```
Enter a positive integer: 100
The sum of the first 100 integers is 5050
```

本例与例 4.1 的运行结果是相同的: 当  $i$  的值等于  $n$  时, 循环终止并且执行程序末尾的输出语句。

注意 `while` 循环本身的控制条件是 `true`, 这表示会永远循环下去。这是一种在 `while` 循环内部进行控制的标准方法。

在循环内部使用 `break` 语句的好处之一是不需要继续执行循环体中其余的语句, 就可以使循环立即中止。

### 例 4.5 Fibonacci 数列

Fibonacci 数列  $F_0, F_1, F_2, F_3, \dots$  是由下面的等式递归定义的:

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \end{cases}$$

例如, 在第三个等式中设  $n = 2$ , 则

$$F_2 = F_{2-1} + F_{2-2} = F_1 + F_0 = 0 + 1 = 1$$

同样地, 当  $n = 3$ , 则

$$F_3 = F_{3-1} + F_{3-2} = F_2 + F_1 = 1 + 1 = 2$$

当  $n = 4$ , 则

$$F_4 = F_{4-1} + F_{4-2} = F_3 + F_2 = 2 + 1 = 3$$

前10个 Fibonacci 数字如图 4.3 所示。

本例打印出不超过输入数字的所有的 Fibonacci 数字:

$n$	$F_n$
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	35

图 4.3 斐波纳契数列 (Fibonacci)

```
int main ()
{ long bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  cout << "Fibonacci numbers < " << bound << ": \n0, 1";
  long f0 = 0, f1 = 1;
  while (true)
  { long f2 = f0 + f1;
    if (f2 > bound) break; // 立即终止循环
    cout << ", " << f2;
    f0 = f1;
    f1 = f2;
  }
}
```

```
Enter a positive integer: 1000
Fibonacci numbers < 1000:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987
```

while 循环中包括一个有 5 个语句的语句块。当条件 ( $f2 > bound$ ) 的值为真时, 执行 break 语句, 立即中止循环, 而没有执行该次循环中剩下的 3 个语句。

注意在字符串 “: \n0, 1” 中换行符 “\n” 的用法。这个字符串在当前行的行尾打印冒号 “:”, 然后, 在下--行的行首打印出 0, 1。

#### 例 4.6 使用 exit ( 0 ) 函数

exit ( ) 函数提供了另一种终止循环的方法。当该函数执行时, 可以中止程序。

```
int main ()
{ long bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  cout << "Fibonacci numbers < " << bound << ": \n0, 1";
  long f0 = 0, f1 = 1;
```

```

while (true)
{
    int f2 = f0 + f1;
    if (f2 > bound) exit (0);    //立即终止程序
    cout << ", " << f2;
    f0 = f1;
    f1 = f2;
}

```

```

Enter a positive integer: 1000
Fibonacci numbers < 1000:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987

```

由于程序中在循环后面没有其他的语句，终止循环就相当于终止程序，因此，该程序与例 4.5 的运行结果相同。

例 4.6 说明一种跳出无限循环的方法。下面的例子说明了怎样中断一个无限循环，但是最好的方法是使用 break 语句，如例 4.20 所示。

#### 例 4.7 中断无限循环

如果没有某些终止的方法，循环将永远执行下去，为了中断它的执行，可以按 <Ctrl> + C (即按住 Ctrl 键的同时按下 C 键)。

```

int main ()
{
    long bound;
    cout << "Enter a positive integer: ";
    cin >> bound;
    cout << "Fibonacci numbers < " << bound << ": \n0, 1";
    long f0 = 0, f1 = 1;
    while (true)        // 错误：无限循环    (按 <Ctrl> + C.)
    {
        long f2 = f0 + f1;
        cout << ", " << f2;
        f0 = f1;
        f1 = f2;
    }
}

```

```

Enter a positive integer: 1000
Fibonacci numbers < 1000:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597
81, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 5
040, 1346269, 2178309, 3524578, 5702887, 9227465, 14930352, 24157817,
63245986, 102334155, 165580141, 267914296, 433494437, 701408733, 11349

```

由于程序中在循环后面没有其他的语句，终止循环就相当于终止程序，因此，该程序与例 4.5 的运行结果相同。

### 4.3 do...while 循环

do...while 语句的语法为：

do 语句 while (条件);

其中, 条件是一个整型表达式, 语句是任意的可执行语句。它重复执行语句, 然后判断条件, 直到条件的值为 false。

do...while 语句的作用与 while 语句基本相同, 差别是它不是在循环的开始判断条件, 而是在循环的最后判断条件, 这意味着可以在循环内部定义循环控制变量而不是在循环之前定义, 还意味着不管控制条件的值是真是假, do...while 循环至少要执行一次。

#### 例 4.8 使用 do...while 循环计算连续整数的和

本例与例 4.1 的运行结果相同。

```
int main ()
{
    int n, i = 0;
    cout << "Enter a positive integer: ";
    cin >> n;
    long sum = 0;
    do
        sum += i++;
    while (i <= n);
    cout << "The sum of the first " << n << " integers is " << sum;
}
```

#### 例 4.9 数的阶乘

数的阶乘  $0!$ ,  $1!$ ,  $2!$ ,  $3!$  ... 是由以下公式递归定义的。

$$\begin{cases} 0! = 1 \\ n! = n(n-1) \end{cases}$$

例如, 设在等式 2 中  $n = 1$ , 则

$$1! = 1((1-1)!) = 1(0!) = 1(1) = 1$$

同样地, 当  $n = 3$ , 则

$$2! = 2((2-1)!) = 2(1!) = 2(1) = 2$$

当  $n = 4$ , 则

$$3! = 3((3-1)!) = 3(2!) = 3(2) = 6$$

前七个数的阶乘如图 4.4 所示。

本例打印出不超过输入数字的所有的数的阶乘。

$n$	$n!$
0	1
1	1
2	2
3	6
4	24
5	120
6	720

图 4.4  $n$  的阶乘

```
int main ()
{
    long bound;
    cout << "Enter a positive integer: ";
    cin >> bound;
    cout << "Factorial numbers < " << bound << ": \n1, 1";
    long f = 1, i = 1;
    do
        f *= ++i;
    while (i <= bound);
}
```

```

    cout << ", " << f;
    |
    while (f < bound);
    |
Enter a positive integer: 1000000
Factorial numbers < 1000000:
1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880

```

do..while 循环重复执行，直到控制条件 ( $f < \text{bound}$ ) 为假。

## 4.4 for 语句

for 语句的语法为：

**for** (初始化；条件；修改) 语句；

其中，初始化、条件和修改是可选的表达式，语句是任意的可执行语句。这三个部分 (初始化；条件；修改) 控制循环。初始化表达式的作用是声明且/或初始化循环控制变量 (s)，它首先被计算；条件表达式的作用是决定循环是否继续执行，它在初始化后立即被计算，如果条件为真，则执行语句；修改的作用是修改控制变量 (s)，它是在执行语句之后被计算的。因此，循环的执行序列为：

- (1) 计算初始化表达式；
- (2) 如果条件表达式的值为假，终止循环；
- (3) 执行语句；
- (4) 计算修改表达式；
- (5) 重复步骤(2)~(4)。

### 例 4.10 使用 for 循环计算连续整数的和

本例与例 4.1 的运行结果相同：

```

int main ()
| int n;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum = 0;
  for (int i = 1; i <= n; i++)
    sum += i;
  cout << "The sum of the first " << n << " integers is " << sum;
|

```

这里，初始化表达式为  $\text{int } i = 1$ ，条件表达式为  $i \leq n$ ，修改表达式为  $i++$ 。注意在例 4.1、例 4.4 和例 4.8 中使用的是相同的表达式。

在标准的 C++ 中，当一个循环控制变量在 for 循环内部声明时，就如例 4.10 中的  $i$ ，它的使用范围就被限制在 for 循环内部，这意味着它不能在 for 循环之外使用，同样还意味着



在 for 循环之外其他的变量可使用与它相同的名字。

#### 例 4.11 再次使用 for 循环控制变量的名字

本例与例 4.1 的运行结果相同。

```
int main ()
{
    int n;
    cout << "Enter a positive integer: ";
    cin >> n;
    long sum = 0;
    for (int i = 1; i < n/2; i++) //这个 i 的范围是这个循环
        sum += i;
    for (int i = n/2; i <= n; i++) //这个 i 的范围是这个循环
        sum += i;
    cout << "The sum of the first " << n << " integers is "
         << sum << endl;
}
```

在本例中的两个 for 循环与在例 4.10 中的一个 for 循环所做的计算相同。它们只是将工作分为两个，在第一个循环中做前面的  $n/2$  的累加，而在第二个循环中做剩下的累加。每个循环都独立地声明了自己的控制变量  $i$ 。

警告：许多非标准的 C++ 编译器将 for 循环控制变量的作用域范围扩展到循环之外。

#### 例 4.12 再次计算数的阶乘

本例与例 4.9 的运行结果相同：

```
int main ()
{
    long bound;
    cout << "Enter a positive integer: ";
    cin >> bound;
    cout << "Factorial numbers that are <= " << bound << ":    1, 1";
    long f = 1;
    for (int i = 2; i <= bound; i++)
        f *= i;
    cout << ", " << f;
}
```

```
Enter a positive integer: 1000000
Factorial numbers < 1000000:
1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880
```

这个 for 循环程序与 do...while 循环程序的运行结果相同，因为它执行相同的指令。在把  $f$  初始化为 1 后，两个程序都将  $i$  初始化为 2，然后，重复执行下面的 5 条语句：打印  $f$ ，将  $f$  与  $i$  相乘， $i$  加 1，检查条件 ( $f \leq \text{bound}$ )，并且当条件为假时终止循环。

正如下例所示，for 语句是非常灵活的。

**例 4.13 使用递减的循环**

本例以倒序打印前十个正整数。

```
int main ()
{ for (int i=10; i > 0; i-- )
    cout << " " << i;
}
```

10 9 8 7 6 5 4 3 2 1

**例 4.14 使用步长大于 1 的 for 循环**

本例判断一个输入的数字是否为素数。

```
int main ()
{ long n;
  cout << "Enter a positive integer: ";
  cin >> n;
  if (n < 2) cout << n << " is not prime." << endl;
  else if (n < 4) cout << n << " is prime." << endl;
  else if (n%2 == 0) cout << n << " = 2*" << n/2 << endl;
  else
  { for (int d=3; d <= n/2; d += 2)
    { if (n%d == 0)
      { cout << n << " = " << d << "*" << n/d << endl;
        exit (0);
      }
    }
    cout << n << " is prime." << endl;
  };
}
```

Enter a positive integer: 101  
101 is prime.

Enter a positive integer: 975313579  
975313579 = 17 \* 57371387.

注意，这个 for 循环的控制变量 i 使用了增量 2。

**例 4.15 使用标记控制 for 循环**

本例在输入的数字序列中找到最大值。

```
int main ()
{ int n, max;
  cout << "Enter positive integers (0 to quit): ";
  cin >> n;
  for (max = n; n > 0; )
  { if (n > max) max = n;
    cin >> n;
  }
  cout << "max = " << max << endl;
}
```

```
Enter positive integers: (0 to quit): 44 77 55 22 99 33 11 66 88 0
max = 99
```

这里的 for 循环由输入变量  $n$  控制；它被循环执行，直到  $n \leq 0$ 。像这样一个控制循环的输入变量被称为标记。

注意在这个 for 循环中的控制机制 ( $\text{max} = n; n > 0;$ )，它没有修改表达式，并且初始化表达式  $\text{max} = n$  没有声明，在 for 循环之前没有声明  $\text{max}$ ，因为在 for 语句块之外，在程序的最后一条输出语句中也使用了  $\text{max}$ 。

#### 例 4.16 使用循环不变式证明 for 循环的正确性

本例求输入的数字序列中的最小值，它与例 4.15 相似。

```
int main ()
{ int n, min;
  cout << "Enter positive integers (0 to quit): ";
  cin >> n;
  for (min = n; n > 0; )
  { if (n < min) min = n;
    // 不变量: 对所有的 n, min ≤ n. 并且 min 等于其中的一个 n
    cin >> n;
  }
  cout << "min = " << min << endl;
}
```

```
Enter positive integers: (0 to quit): 44 77 55 22 99 33 11 66 88 0
min = 11
```

在 for 循环的语句块中的注释被称为循环不变式。它说明了具有以下两个特征属性的条件：(1) 在每次循环中它都为真；(2) 当循环终止时它为真证明了循环的正确执行。在这种情况下，条件  $\text{min} \leq n$  for all  $n$  总是正确的，因为前面的 if 语句在最后输入的  $n$  值小于  $\text{min}$  的上一值时会改变  $\text{min}$  的值。并且条件  $\text{min}$  equals one of the  $n$  总是为真的，因为  $\text{min}$  最初被赋值为第一个  $n$ ，而且只有当  $\text{min}$  被赋值为一个新输入的  $n$  值时它的值才改变。最后，当循环终止时条件为真意味着  $\text{min}$  是所有输入数中的最小值，并且输出恰好是循环的目的。

#### 例 4.17 在一个 for 循环中使用多个控制变量

本例中的 for 循环使用了两个控制变量。

```
int main ()
{ for (int m = 95, n = 11; m % n > 0; m -= 3, n++)
  cout << m << " % " << n << " = " << m % n << endl;
}
```

```
95 % 11 = 7
92 % 12 = 8
89 % 13 = 11
86 % 14 = 2
83 % 15 = 8
```

两个控制变量  $m$  和  $n$  的值是在 `for` 循环的控制机制中被声明和初始化的，然后，在每次循环中  $m$  每次递减 3 而  $n$  递增 1，产生了  $(m, n)$  对的序列  $(95, 11)$ ,  $(92, 12)$ ,  $(89, 13)$ ,  $(86, 14)$ ,  $(83, 15)$ ,  $(80, 16)$ 。因为 16 可除尽 80，所以循环终止于数对  $(80, 16)$ 。

#### 例 4.18 循环的嵌套

本例打印一个乘法表。

```
#include <iomanip> // 定义 setw ()
#include <iostream> // 定义 cout
using namespace std;
int main ()
{ for (int x=1; x <= 12; x++)
  { for (int y=1; y <= 12; y++)
    cout << setw (4) << x*y;
    cout << endl;
  }
}
```

1	2	3	4	5	6	7	8	9	10	11	12
2	4	6	8	10	12	14	16	18	20	22	24
3	6	9	12	15	18	21	24	27	30	33	36
4	8	12	16	20	24	28	32	36	40	44	48
5	10	15	20	25	30	35	40	45	50	55	60
6	12	18	24	30	36	42	48	54	60	66	72
7	14	21	28	35	42	49	56	63	70	77	84
8	16	24	32	40	48	56	64	72	80	88	96
9	18	27	36	45	54	63	72	81	90	99	108
10	20	30	40	50	60	70	80	90	100	110	120
11	22	33	44	55	66	77	88	99	110	121	132
12	24	36	48	60	72	84	96	108	120	132	144

在外层  $x$  循环每次执行时打印乘法表的一行。例如，在第一次循环中，当  $x = 1$  时，内层  $y$  重复执行 12 次，对从 1 到 12 的每个  $y$  打印  $1 * y$ 。然后，外层  $x$  循环在  $x = 2$  时执行第二次循环，内层的  $y$  再次重复执行 12 次，这次对从 1 到 12 的每个  $y$  打印  $2 * y$ 。注意“`cout << endl;`”语句必须放在外层循环之内和内层循环之外，以便在每次外层循环执行后换行。

本程序中使用了流控制符 `setw` 来对每个整数设置输出区域的宽度。表达式 `setw (4)` 表示对下一个输出“设置输出区域的宽度为 4 列”，这样就将输出排列为按整数右边对齐的 12 列的表格。流控制符是在头文件 `<iomanip>` 中定义的，因此，在本程序开头必须包含伪指令

```
#include <iomanip>
```

另外，还要包含头文件 `<iostream>`。

#### 例 4.19 测试循环不变式

本例输入一个数字，计算并打印出以 2 为底的离散对数（最大的整数  $\leq$  该数字的以 2 为底的对数）。它通过打印每次循环相应的值测试循环不变式：

```

#include <cmath>      // 定义 pow() 和 log()
#include <iostream>    // 定义 cin 和 cout
#include <iomanip>      // 定义 setw()
using namespace std;

int main()
{ long n;
  cout << "Enter a positive integer: ";
  cin >> n;
  int d=0;    // 以 2 为底 n 的离散对数值
  double p2d=1; // = 2^d
  for (int i=n; i>1; i/=2, d++)
  { // 不变量:  $2^d \leq n/i < 2 \times 2^d$ 
    p2d=pow(2, d); // = 2^d
    cout << setw(2) << p2d << " <= " << setw(2) << n/i
         << " < " << setw(2) << 2*p2d << endl;
  }
  p2d=pow(2, d); // = 2^d
  cout << setw(2) << p2d << " <= " << setw(2) << n
       << " < " << setw(2) << 2*p2d << endl;
  cout << " The discrete binary logarithm of " << n
       << " is " << d << endl;
  double lgn = log(n)/log(2); // 以 2 为底 n 的对数值
  cout << "The continuous binary logarithm of " << n
       << " is " << lgn << endl;
}

```

```

Enter a positive integer: 63
1 <= 1 < 2
2 <= 2 < 4
4 <= 4 < 8
8 <= 9 < 16
16 <= 21 < 32
32 <= 63 < 64
The discrete binary logarithm of 63 is 5
The continuous binary logarithm of 63 is 5.97728

```

以 2 为底的离散对数是输入的数字在达到 1 之前能被 2 整除的次数。因此，for 循环将 i 初始化为 n，然后，在每次循环时将 i 除以 2。计数器 c 计算循环的次数。所以当循环终止时，c 中的值就是 n 的以 2 为底的离散对数。

除了使用了在头文件 <iomanip> 中定义的 setw() 函数外，本程序中还使用了在头文件 <cmath> 中定义的函数 log()，这个函数返回 n 的自然对数（以 e 为底）： $\log(n) = \log_e n = \ln n$ 。表达式  $\log(n)/\log(2)$  被用来计算 n 的以 2 为底的对数： $\log_2 n = \lg n = (\ln n)/(\ln 2)$ 。打印的结果将以 2 为底的离散对数与以 2 为底的连续对数相比较，前者与后者小的最大整数相等（该数的基数）。

在本例中的循环不变式是条件  $2^d \leq n/i < 2 \times 2^d$ （即  $2^d \leq n/i < 2 \cdot 2^d$ ）。它是通过打印 3 个表达式 p2d、n 和  $2 \times p2d$  来测试的，其中 p2d 是用定义在头文件 <cmath> 中的幂函数 pow() 来计算的。

可以证明本例中的 for 循环总能正确地计算出以 2 为底的离散对数。在开始时,  $d = 0$  并且  $i = n$ , 所以  $2^d = 2^0 = 1$ ,  $n/i = n/n = 1$ , 并且  $2 \cdot 2^d = 2 \cdot 1 = 2$ 。由此  $2^d \leq n/i < 2 \cdot 2^d$ 。在每次循环中,  $d$  递增并且  $i$  被 2 等分, 所以  $n/i$  被加倍。这样条件  $2^d \leq n/i < 2 \cdot 2^d$  保持不变性; 即, 它开始时是正确的, 并且在整个循环周期中始终保持正确性。当循环终止时,  $i = 1$ , 所以, 条件变成  $2^d \leq n/1 < 2 \cdot 2^d$ , 这与  $2^d \leq n < 2^{d+1}$  等价。该表达式的对数为  $d = \lg(2^d) \leq \lg n < \lg(2^{d+1}) = d + 1$ , 所以  $d$  总是  $\leq \lg n$  的最大整数。

## 4.5 break 语句

前面已经看到 break 语句用于 switch 语句中的情况, 它也可以用于循环中。当 break 执行时, 会终止循环, 在该点“跳出”循环。

### 例 4.20 使用 break 语句终止循环

本例与例 4.1 的运行结果相同, 它使用 break 语句控制循环:

```
int main ()
{ int n, i=1;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  while (true)
  { if (i > n) break;
    sum += i++;
  }
  cout << "The sum of the first " << n << " integers is " << sum;
}
```

Enter a positive integer: 8  
The sum of the first 8 integers is 36

像例 4.1 一样, 当 ( $i \leq n$ ) 时, 循环将继续执行。但是当  $i > n$  时, 则执行 break 语句, 立即终止循环。

break 语句提供了更加灵活的循环控制方式。通常 while 循环、do...while 循环或者 for 循环只能在循环块中的整个语句序列的开始或结尾处终止, 但是 break 语句可以任意放在循环中的其他语句之中, 所以它可以用来在循环块的任意位置终止循环。以下例子说明了这一点。

### 例 4.21 使用标记控制输入

本例读入一个整数的序列, 当输入为 0 时终止, 并且打印出它们的平均值。

```
int main ()
{ //计算输入数的平均值
  int n, count=0, sum=0;
  cout << "Enter positive integers (0 to quit):" << endl;
  for (;;) //“永远”
```

```

    cout << "\t" << count + 1 << ": ";
    cin >> n;
    if (n <= 0) break;
    ++count;
    sum += n;
}
cout << "The average of those " << count << " positive numbers is "
    << float (sum) /count << endl;
}
Enter positive integers (0 to quit):
    1: 4
    2: 7
    3: 1
    4: 5
    5: 2
    6: 0
The average of those 5 positive numbers is 3.8

```

当0被输入时，执行 break 语句，立即终止 for 循环并且转去执行最后的输出语句。如果没有 break 语句，语句 ++count 就必须是有条件的，或者 count 必须在循环之外递减或初始化为 -1。

注意在 for 循环的控制机制中的三部分均为空：for (;;)，这种结构表示“永远”，如果没有 break 语句，该循环将变为无限循环。

当 break 语句被用于循环嵌套中时，仅对它所在的那层循环起作用；外层循环将继续执行，不会受 break 影响。以下的例子说明了这一点。

#### 例 4.22 在循环嵌套中使用 break

既然乘法满足交换性（即， $3 \times 4 = 4 \times 3$ ），乘法表中主对角线以上的数字经常被删去。本例修改了例 4.18，打印出一个三角形的乘法表。

```

int main ()
{ for (int x=1; x <= 12; x++)
  { for (int y=1; y <= 12; y++)
    { if (y > x) break;
      else cout << setw (4) << x*y;
    }
    cout << endl;
  }
}

```

```

1
2  4
3  6  9
4  8 12 16
5 10 15 20 25
6 12 18 24 30 36
7 14 21 28 35 42 49
8 16 24 32 40 48 56 64
9 18 27 36 45 54 63 72 81
10 20 30 40 50 60 70 80 90 100
11 22 33 44 55 66 77 88 99 110 121
12 24 36 48 60 72 84 96 108 120 132 144

```

当  $y > x$  时, 执行内层的  $y$  循环并且开始下一次的外层  $x$  循环。例如, 当  $x = 3$ ,  $y$  循环重复执行 3 次 ( $y = 1, 2, 3$ ), 打印出 3 6 9。然后在第 4 次循环中, 条件 ( $y > x$ ) 为真, 因此执行 `break` 语句, 将控制立即转去执行 `cout << endl` 语句 (在内层  $y$  循环之外), 然后, 外层  $x$  循环开始第 4 次循环, 此时  $x = 4$ 。

## 4.6 continue 语句

`break` 语句跳过循环块中剩余的语句, 立即跳到循环外面的下一个语句处。`continue` 语句与 `break` 类似, 它也跳过循环块中剩余的语句, 但它不是终止循环, 而是转去执行下一次循环。它是在跳过本次循环的剩余语句后继续循环的。

### 例 4.23 使用 `continue` 和 `break` 语句

本例说明 `continue` 和 `break` 语句的区别:

```
int main ()
{ int n;
  for (;;)
  { cout << "Enter int: ";  cin >> n;
    if (n%2 == 0) continue;
    if (n%3 == 0) break;
    cout << "\tBottom of loop.\n";

    cout << "\tOutside of loop.\n";
  }
}
```

```
Enter int : 7
    Bottom of loop.
Enter int : 4
Enter int : 9
    Outside of loop.
```

当  $n$  的值为 7 时, 两个 `if` 语句的条件都为假并且程序的控制转到循环的底部。当  $n$  的值为 4 时, 第一个 `if` 语句的条件为真 (4 是 2 的倍数), 所以, 程序的控制跳过循环剩余的语句并且立即跳转到循环的顶部继续执行下一次循环。当  $n$  的值为 9 时, 第一个 `if` 语句的条件为假 (9 不是 2 的倍数), 但是第二个 `if` 语句的条件为真 (9 是 3 的倍数), 所以程序的控制跳出循环, 并且立即跳到循环后面的第一条语句处。

## 4.7 goto 语句

`break` 语句、`continue` 语句和 `switch` 语句都会导致程序控制分支转到非正常执行的地方, 程序分支所转去的目的是这样决定的: `break` 转到循环外的下一条语句处, `continue` 转到循环的继续条件处, 而 `switch` 转到正确的 `case` 子句处。以上这三个语句都称为跳转语句, 因为它们都导致程序“跳转”到其他的语句处。



goto 语句是另一种跳转语句，它所转去的目的是由语句中的标号规定的。

标号是紧跟着一个冒号的简单标识符，放在一个语句的前头。标号的作用就像在 switch 语句中的 case 语句：它们指定跳转的目的。

例 4.22 中说明了通常在循环嵌套中 break 语句是如何作用的：只跳出包含有 break 语句的最内层循环。正如一下例子所说明的，要跳出几层循环或所有嵌套的循环就需要用到 goto 语句。

#### 例 4.24 使用 goto 语句跳出循环嵌套

```
int main ()
{
    const int N=5;
    for (int i=0; i < N; i++)
        for (int j=0; j < N; j++)
            for (int k=0; k < N; k++)
                if (i+j+k > N) goto esc;
                else cout << i-j+k << " ";
                cout << " * ";
            }
    esc: cout << "." << endl;    // 在 i 循环内，在 j 循环外
}

0 1 2 3 4 * 1 2 3 4 5 * 2 3 4 5
1 2 3 4 5 * 2 3 4 5
2 3 4 5
3 4 5
4 5
```

当执行到最内层的 k 循环中所包含的 goto 语句时，程序跳转到最外层的 i 循环底部被标号所标识的语句处。由于它是 i 循环的最后一条语句，所以 i 循环将在执行该语句后开始下一次循环。

当 i 和 j 为 0 时，k 循环重复执行 5 次，打印出 0 1 2 3 4，后面跟一个星号 \*，然后 j 递增到 1 并且 k 循环又重复执行 5 次，打印出 1 2 3 4 5，后面跟一个星号 \*，然后 j 递增到 2 并且 k 循环重复执行 4 次，打印出 2 3 4 5。但是下面在 k 的下次循环时，i = 0，j = 2，并且 k = 4，因此 i+j+k = 6，使得 goto 语句第一次执行，所以程序立即跳转到标号所标出的语句处，打印一个圆点然后继续执行下一行。注意 k 循环和 j 循环在完成它们的循环前都被终止。

现在 i = 1 并且中层的 j 循环从 j = 1 又开始循环。k 循环重复执行 5 次，打印出 1 2 3 4 5 后跟一个星号 \*，然后 j 递增到 1 并且 k 循环重复执行 4 次，打印出 2 3 4 5。但是下面在 k 的下次循环时，i = 1，j = 2，并且 k = 3，因此 i+j+k = 6，使得 goto 语句第二次执行，所以程序再次立即跳转到标号所标出的语句处，打印一个圆点然后继续执行下一行。

在外层 i 循环的三个循环序列中，内层的 k 循环从没有完成所有的循环，因为 i+j+4

总是比 5 大（因为  $i$  大于 2），所以没有再打印出星号 \*。

注意，标号标明的语句可以放在循环中的任意位置，甚至可以放在所有循环之外。在后一种情况下，`goto` 语句将终止嵌套中的所有 3 个循环。

还要注意标号语句是如何设计的。习惯是将它放在锯齿状的左边而使它更加醒目。如果不是一个标号语句，可以设计为：

```
|
| cout << "." << endl;
|
```

而不是：

```
|
| esc: cout << "." << endl;
|
```

例 4.24 指出了跳出循环嵌套的一种方法，另一种方法是使用标志。标志是一个初始化为 `false` 的布尔变量，并且在后面被设置成 `true` 以作为一个意外事件发生的信号；通常当标志变为 `true` 时，程序执行将被中断。以下例子说明了这一点。

#### 例 4.25 使用标志跳出循环嵌套

本例与例 4.24 的运行结果相同：

```
int main ()
| const int N= 5;
| bool done= false;
| for (int i=0; i < N; i++)
|   for (int j=0; j < N && ! done; j++)
|     for (int k=0; k < N && ! done; k++)
|       if (i+j+k > N) done = true;
|       else cout << i+j+k << " ";
|       cout << " * ";
|   |
|   cout << "." << endl; // 在 i 循环内，在 j 循环外
|   done = false;
|
```

当标志 `done` 变为真时，内层的  $k$  循环和中间层的  $j$  循环都将被终止，并且外层的  $i$  循环将在打印出圆点后结束它的本次循环而开始下一行，重新设置 `done` 标志为 `false`，然后开始下一次循环，同例 4.24。

## 4.8 生成伪随机数

计算机最重要的应用之一是对现实世界的系统进行仿真。许多高新技术的研究和开发都

极大地依赖于该技术，目的是可以不真正地与现实世界的系统直接交互而对这些系统进行研究。

仿真需要计算机生成伪随机数来模拟现实世界的不确定性。当然，计算机不能真正生成真实的随机数，因为计算机是具有确定性的：给定相同的输入，同一个计算机总是会产生相同的输出。但是它可能生成看来是随机产生的数字；即，数字在一个给定的整数中均匀分布，并且没有可辨别的模式。这样的数字称为伪随机数。

标准 C 的头文件 `<cstdlib>` 定义了函数 `rand()` 来在 0 到 `RAND_MAX` 范围之间生成伪随机整数，其中 `RAND_MAX` 是一个也在 `<cstdlib>` 中定义的常量。每次调用 `rand()` 函数时，它都会在该范围内生成另一个 unsigned 整数。

#### 例 4.26 生成伪随机数

本例使用了 `rand()` 函数来生成伪随机数：

```
#include <cstdlib> // 定义 rand() 函数和常量 RAND_MAX
#include <iostream>
using namespace std;
```

```
int main ()
| // 打印伪随机数
  for (int i = 0; i < 8; i++)
    cout << rand() << endl;
  cout << "RAND_MAX = " << RAND_MAX << endl;
|
```

```
1103527590
377401575
662824084
1147902781
2035015474
368800899
1508029952
486256185
RAND_MAX = 2147483647
```

```
1103527590
377401575
662824084
1147902781
2035015474
368800899
1508029952
486256185
RAND_MAX = 2147483647
```

在每次运行时，计算机生成 8 个 unsigned 整数，它们均匀分布在整数 0 到 `RAND_MAX` 之间，这里 `RAND_MAX` 在计算机上为 2 147 483 647。遗憾的是每次运算都产生相同的数字序列。这是因为它们是由同一个“种子”生成的。

每个伪随机数都是通过在内部定义的一个特殊的“数字压碎”函数由上一个伪随机数生

成的。第一个伪随机数是由一个内部定义的变量生成的，该变量称为这个序列的种子。在默认情况下，在程序每次运行时种子由计算机初始化为同一个值。为了避免这种破坏随机性的现象，可以使用 `srand()` 函数来选择自己的种子。

#### 例 4.27 交互地设置种子

本例除了允许交互地设置生成伪随机数的种子，基本与例 4.26 相同：

```
#include <cstdlib> // 定义 rand() 函数和 srand() 函数
#include <iostream>
using namespace std;

int main()
{ // 打印伪随机数
    unsigned seed;
    cout << "Enter seed: ";
    cin >> seed;
    srand(seed); // 初始化种子
    for (int i = 0; i < 8; i++)
        cout << rand() << endl;
```

```
Enter seed : 0
12345
1406932606
654583775
1449466924
229283573
1109335178
1051550459
1293799192
```

```
Enter seed : 1
1103527590
377401575
662824084
1147902781
2035015474
368800899
1508029952
486256185
```

```
Enter seed : 12345
1406932606
654583775
1449466924
229283573
1109335178
1051550459
1293799192
794471793
```

代码 `srand(seed)` 将变量 `seed` 的值赋给内部的“种子”，函数 `rand()` 使用该种子来初

始化它所生成的伪随机数序列。不同的种子会生成不同的结果。

注意在第三次运行时使用的 seed 的值 12345 是由 rand () 在第一次运行时生成的, 相应地第三次运行中生成的前 7 个数字与第一次运行时生成的从第 2 个开始的 7 个数字相同。还要注意在第二次运行时生成的数字序列与例 4.26 中的相同, 这个现象说明, 在计算机中, 默认的种子值为 1。

必须输入一个 seed 的值的这个问题可以使用计算机的系统时钟来解决。系统时钟以秒为单位跟踪当前的时间。在头文件 <ctime> 中定义的函数 time () 以一个 unsigned 整数返回当前时间, 可以作为 rand () 函数的种子。

#### 例 4.28 利用系统时钟设置种子

本例除了根据系统时钟设置伪随机数的种子, 基本与例 4.27 相同。

注意: 如果你的编译器不能识别头文件 <ctime>, 那么可以使用头文件 <time.h> 来代替。

```
#include <cstdlib> //定义 rand () 和 srand () 函数
#include <ctime>    // 定义 time () 函数
#include <iostream>
// #include <time.h> // 如果不认识 <ctime> 使用这个
using namespace std;
int main ()
{ // prints pseudo-random numbers:
    unsigned seed = time (NULL); // 使用系统时钟
    cout << "seed = " << seed << endl;
    srand (seed); // 初始化种子
    for (int i = 0; i < 8; i++)
        cout << rand () << endl;
}
```

这里是使用 Motorola 处理器的 UNIX 工作站的两次运行情况。

```
seed = 808148157
1877361330
352899587
1443923328
1857423289
200398846
1379699551
1622702508
715548277
```

```
seed = 808148160
892939769
1559273790
1468644255
952730860
1322627253
1305580362
844657339
440402904
```

在第一次运行时, `time()` 函数返回整数 808 148 157, 被用做随机数字生成器的“种子”。第二次运行在 3 秒之后进行, 所以 `time()` 函数返回整数 808 148 160 来生成一个完全不同的序列。

这里是使用 Intel 处理器的 Windows PC 机的两次运行情况:

```
seed = 943364015
2948
15841
72
25506
30808
29709
13115
2525
```

```
seed = 943364119
17427
20464
13149
5702
12766
1424
16612
31746
```

在许多仿真程序中, 生成随机整数的种子是均匀地分布在一个给定的范围中的, 以下的例子说明了怎样去做。

#### 例 4.29 在一个给定范围内生成伪随机数

本例除了是在限定的范围内生成伪随机数之外, 基本与例 4.28 相同:

```
#include <cstdlib>
#include <ctime>      // 定义 time() 函数
#include <iostream>
// #include <time.h>   // 如果不认识 <ctime> 使用这个
using namespace std;

int main()
{ // prints pseudo-random numbers:
  unsigned seed = time(NULL);      // 使用系统时钟
  cout << "seed = " << seed << endl;
  srand(seed);                     // 初始化种子
  int min, max;
  cout << "Enter minimum and maximum: ";
  cin >> min >> max;                // 最小和最大数
  int range = max - min + 1;        // range 中的数字数
  for (int i = 0; i < 20; i++)
  { int r = rand() / 100 % range + min;
    cout << r << " ";
  }
  cout << endl;
```

这里有两次运行的情况：

```
seed = 808237677
Enter minimum and maximum: 1 100
85 57 1 10 5 73 81 43 46 42 17 44 48 9 3 74 41 4 30 68
seed = 808238101
Enter minimum and maximum: 22 66
63 29 56 22 53 57 39 56 43 36 62 30 41 57 26 61 59 26 28
```

第一次运行时生成 20 个在 1 到 100 之间均匀分布的整数，第二次运行时生成 20 个在 22 到 66 之间均匀分布的整数。

在 for 循环中，将 `rand()` 除以 100 来生成随机数中两侧的数字。这样可以解决这个特殊的随机数生成器必须在奇数和偶数间选择来生成数字的问题。然后 `rand() / 100 % range` 产生在 0 到 `range - 1` 范围内的随机数，而 `rand() / 100 % range + min` 产生在 `min` 到 `max` 范围之间的随机数。

## 复 习 题

- 4.1 当开始时控制条件为假（即 0），while 循环将如何执行？
- 4.2 在 for 循环中何时控制变量可以在循环前声明（而不是在控制机制内部）？
- 4.3 break 语句如何提供更好的循环控制？
- 4.4 以下语句重复执行的最少次数是多少？
  - a. while 循环
  - b. do...while 循环
- 4.5 以下循环有多少错误：

```
while(n <= 100)
    sum += r * n;
```

- 4.6 如果 `s` 是一个复合条件，并且 `e1`、`e2` 和 `e3` 是表达式，则以下的程序片断有什么区别？

```
for (e1; e2; e3)
    s;

e1;
while (e2)
    { s;
      e3
    };
```

- 4.7 以下程序有多少错误？

```
int main ()
{ const double PI;
```

```
int n;  
PI = 3.14159265358979;  
n = 22;  
|
```

4.8 什么是“无限循环”？它有什么作用？

4.9 如何构造一个循环，使得它可以使用一个语句在语句块中间终止循环？

4.10 为什么要避免测试浮点数的相等性？

## 习 题

4.1 跟踪以下代码片断，说明每个变量在每次变化后的值。

```
float x = 4.15;  
for (int i = 0; i < 3; i++)  
    x *= 2;
```

4.2 假设  $e$  是一个表达式并且  $s$  是一个语句，将以下的每个 for 循环改写成等价的 while 循环。

```
a. for (; e;) s  
b. for (; ; e) s
```

4.3 将以下的 for 循环改写成 while 循环：

```
for (int i = 1; i <= n; i++)  
    cout << i * i << " ";
```

4.4 描述以下程序的输出：

```
int main ()  
{ for (int i = 0; i < 8; i++)  
    if (i % 2 == 0) cout << i + 1 << " \t";  
    else if (i % 3 == 0) cout << i * i << " \t";  
    else if (i % 5 == 0) cout << 2 * i - 1 << " \t";  
    else cout << i << " \t";  
}
```

4.5 描述以下程序的输出：

```
int main ()  
{ for (int i = 0; i < 8; i++)  
    if (i % 2 == 0) cout << i + 1 << endl;  
    else if (i % 3 == 0) continue;  
    else if (i % 5 == 0) break;  
    cout << "End of program. \n";  
}  
cout << "End of program. \n";  
}
```



- 4.6 在一个 32 位的 float 类型中, 有 23 位用来存放尾数而有 8 位用来存放指数。
- 这个 32 位的 float 类型可以产生的数字精度是多少?
  - 这个 32 位的 float 类型的数量级的范围多少?
- 4.7 写出并运行一个程序, 使用 while 循环计算并打印一个给定的平方数的和。例如, 如果输入 5, 则程序将打印出 55, 它等于  $1^2 + 2^2 + 3^2 + 4^2 + 5^2$ 。
- 4.8 写出并运行一个程序, 使用 for 循环计算并打印一个给定的平方数的和。
- 4.9 写出并运行一个程序, 使用 do...while 循环计算并打印一个给定的平方数的和。
- 4.10 写出并运行一个程序, 直接使用商运算符/和余数运算符%实现正整数的除法。
- 4.11 写出并运行一个程序, 以倒序排列一个给定的正整数的各位数字 (参见习题 3.13)。
- 4.12 使用巴比伦算法计算 2 的平方根。这个算法 (之所以这样叫是应为它是古巴比伦人使用的) 通过  $\sqrt{2}$  反复地用一个更接近的估计值  $(x + 2/x)/2$  代替  $x$  来计算。注意这里只是简单地取  $x$  和  $2/x$  的平均值。
- 4.13 写出一个程序, 求一个给定数字的平方根, 更精确地说是求一个平方根小于或等于这个给定数字的最大整数。
- 4.14 使用欧基里德算法, 求给定的两个整数的最大公约数。这个算法通过反复地用较小整数去除较大的整数, 并且用余数来代替较大的整数, 将一个正整数对  $(m, n)$  转换为数对  $(d, 0)$ 。当余数为 0 时, 在数对中的另一个非 0 的数就是最初的两个数 (也是所有的中间数对) 的最大公约数。例如, 如果  $m$  为 532 且  $n$  为 112, 则欧基里德算法通过下面的过程将数对  $(532, 112)$  减少为  $(28, 0)$ :

$$(532, 112) \rightarrow (112, 84) \rightarrow (84, 28) \rightarrow (28, 0)$$

因此, 28 就是 532 和 112 的最大公约数。这个结果可改为  $532 = 28 \cdot 19$  和  $112 = 28 \cdot 8$ 。欧基里德算法的原理是在序列中的每个数对都有相同的公约数集合, 它们恰好是最大公约数的因子。在上例中, 公约数集合为  $\{1, 2, 4, 7, 14, 28\}$ 。在递减过程中这个公约数集合保持不变的原因是当  $m = n \cdot q + r$ , 一个数字是  $m$  和  $n$  的公约数当且仅当它是  $n$  和  $r$  的公约数。

## 复习题答案

- 4.1 如果一个 while 循环的控制条件最初为假, 则循环整个被跳过; 在循环中的语句根本就不执行。
- 4.2 如果 for 循环中的控制变量在循环的语句块外部使用, 则它必须在循环之前声明 (而不是在它的控制机制中), 正如例 4.14 所示。
- 4.3 break 语句允许在循环块内部的任意语句后立即终止循环, 提供了更好的循环控制。如果没有 break 语句, 循环仅能在语句块的开始或结尾终止。
- 4.4 a. while 循环执行的最少次数是 0 次;  
b. do...while 循环执行的最少次数是 1 次。

- 4.5 这是一个无限循环，因为它的控制变量  $n$  的值没有改变。
- 4.6 除了  $s$  是一个 `break` 语句或是一个包含了 `break` 语句或 `continue` 语句的复合语句（即一个语句块）之外，这两个程序段没有区别。例如，以下的 `for` 语句将重复执行 4 次然后终止：

```
for (i = 0; i < 4; i++)
    if (i == 2) continue;
```

但是以下的 `while` 语句将是一个无限循环：

```
i = 0;
while (i < 4)
{ if (i == 2) continue;
  i++;
}
```

- 4.7 常量 `P1` 没有初始化。每个常量必须在声明的同时初始化。
- 4.8 无限循环是一个失去控制的连续执行；它只能通过循环内部的分支语句（如 `break` 或 `goto` 语句）或者是通过中断程序（例如使用 `Ctrl + C`）来终止。如果无限循环是被分支语句终止的，则该循环是有用的。
- 4.9 循环可以使用一条 `break` 语句或 `goto` 语句在它的语句块中间终止。
- 4.10 浮点变量会产生舍入错。在经过算术变换之后，浮点变量的确切的值也许不是我们所期望的值，所以，诸如  $(y == x)$  的测试可能不能正确地进行。

## 习 题 答 案

- 4.1 首先， $x$  被初始化为 4.15 且  $i$  被初始化为 0，然后  $x$  经过三次 `for` 循环被加倍三次。
- 4.2 与 `while` 循环等价的是：

- a. `while (e) s;`  
 b. `while (true) {s; e;}`，假定其中  $s$  不包含 `break` 或 `continue` 语句

- 4.3 与 `while` 循环等价的是：

```
int i = 1;
while (i <= n)
{ cout << i * i << " ";
  i++;
}
```

- 4.4 输出为：

1      1      3      9      5      9      7      7

- 4.5 输出为：

End of program.

```

End of program.
3
End of program.
5
End of program.
End of program.

```

- 4.6 a. 这 23 位数表示了 24 位的尾数，第一位一定是 1，所以不用存储了，这样就表示了 24 位。这 24 位可以表示  $2^{24}$  个数字，并且  $2^{24} = 16\,777\,216$ ，它的范围有 7 位数字，于是可以完全地表示 7 位数字。但是最后一位由于舍入可能不太精确，这样 32 位的浮点类型可以精确到 6 位数。
- b. 32 位浮点数中用做指数的 8 位可以表示  $2^8 = 256$  个不同的数字，其中的两个保留用来指明下溢和溢出，剩下的 254 个数用做指数。因此，指数的范围从 -126 到 +127，可以表示的范围是从  $2^{-126} = 1.75494 \times 10^{-38}$  到  $2^{127} = 1.70141 \times 10^{38}$ 。
- 4.7 本程序使用了 while 循环来计算第一个 n 的平方，其中 n 是输入的：

```

int main()
{ int n;
  cout << "Enter a positive integer: ";
  cin >> n;
  int sum=0, i=0;
  while (i++ < n)
    sum += i*i;
  cout << "The sum of the first " << n << " squares is "
        << sum << endl;
}

```

```

Enter a positive integer: 6
The sum of the first 6 squares is 91

```

- 4.8 本程序使用了 for 循环来计算第一个 n 的平方，其中 n 是输入的：

```

int main()
{ int n;
  cout << "Enter a positive integer: ";
  cin >> n;
  int sum=0;
  for (int i=1; i <= n; i++)
    sum += i*i;
  cout << "The sum of the first " << n << " squares is "
        << sum << endl;
}

```

- 4.9 本程序使用了 do..while 循环来计算第一个 n 的平方，其中 n 是输入的：

```

int main()
{ int n;
  cout << "Enter a positive integer: ";
  cin >> n;
  int sum=0, i=1;

```

```

do
{
    sum += i*i;
}
while (i++ < n);
cout << "The sum of the first " << n << " squares is "
      << sum << endl;
}

```

Enter a positive integer: 6

The sum of the first 6 squares is 91

- 4.10 本程序直接使用商运算符/和余数运算符%实现正整数的除法，这里的算法是：用分数 $n/d$ ，反复地从 $n$ 中减去 $d$ ，直到 $n$ 小于 $d$ 。此时， $n$ 的值就是余数，循环的次数 $q$ 就是商。

```

int main()
{
    int n, d, q, r;
    cout << "Enter numerator: ";
    cin >> n;
    cout << "Enter denominator: ";
    cin >> d;
    for (q = 0, r = n; r >= d; q++)
        r -= d;
    cout << n << " / " << d << " = " << q << endl;
    cout << n << " % " << d << " = " << r << endl;
    cout << "(" << q << ") (" << d << ") + (" << r << ") = "
          << n << endl;
}

```

```

Enter numerator: 30
Enter denominator: 7
30 / 7 = 4
30 % 7 = 2
(4) (7) + (2) = 30

```

程序循环了4次： $30 - 7 = 23$ ， $23 - 7 = 16$ ， $16 - 7 = 9$ ， $9 - 7 = 2$ 。所以商是4，余数是2。注意对整数的除法，该关系式一定为真：

(商) (分母) + (余数) = (分子)

- 4.11 这里的技巧是将给定的整数一次去掉一位数字并且将它们“聚积”为倒序的整数。

```

int main()
{
    long m, d, n = 0;
    cout << "Enter a positive integer: ";
    cin >> m;

    while (m > 0)
    {
        d = m % 10;    // d 将是 m 的最右边数字
        m /= 10;        // 然后从 m 中去掉那个数字
        n = 10*n + d;   // 并把那个数加到 n 上
    }
    cout << "The reverse is " << n << endl;
}

```

```
Enter a positive integer: 123456
The reverse is 654321
```

运行时,  $m$  的初值为 123 456。在第一次循环中,  $d$  被赋值为数字 6,  $m$  减小为 12 345,  $n$  增加到 6。在第二次循环中,  $d$  被赋值为数字 5,  $m$  减小为 1 234,  $n$  增加到 65。在第三次循环中,  $d$  被赋值为数字 4,  $m$  减小为 123,  $n$  增加到 654。以此类推, 在第六次循环中,  $d$  被赋值为数字 1,  $m$  减小为 0,  $n$  增加到 654 321。

#### 4.12 巴比伦算法的实现为:

```
#include <cmath> // 定义 fabs() 函数
#include <iostream>
using namespace std;
int main()
{ const double TOLERANCE = 5e-8;
  double x = 2.0;
  while (fabs(x*x - 2.0) > TOLERANCE)
  { cout << x << endl;
    x = (x + 2.0/x)/2.0; // x 和 2/x 的平均数
  }
  cout << "x = " << x << ", x*x = " << x*x << endl;
}
```

```
2
1.5
1.41667
1.41422
x = 1.41421, x*x = 2
```

这里使用了  $5e-8$  的“公差” ( $= 0.00000005$ ) 来确保精确到 7 位数字。在头文件 `<cmath>` 中定义的 `fab()` 函数 (求浮点数的绝对值), 返回传递给它的表达式的绝对值, 所以循环重复执行, 至  $x * x$  的值在给定的公差 2 内循环终止。

#### 4.13 本程序求给定数的平方根, 使用了“无遗漏的”算法来求所有小于或等于给定数的正整数的平方根:

```
int main()
{ float x;
  cout << "Enter a positive number: ";
  cin >> x;
  int n = 1;
  while (n*n <= x)
    ++n;
  cout << "The integer square root of " << x << " is "
    << n-1 << endl;
}
```

```
Enter a positive number: 1234.56
The integer square root of 1234.56 is 35
```

开始时  $n = 1$ , 然后  $n$  不断递增直到  $n * n > x$ 。当 for 循环终止时,  $n$  是平方大于  $x$

的最小整数，所以  $n-1$  是  $x$  的平方根。注意，在 for 循环中使用了空语句。在循环中需要做的每个语句都在循环的控制部分，但是，在循环结束时的分号仍是必需的。

#### 4.14 欧基里德算法的实现如下：

```
int main()
{ int m, n, r;
  cout << "Enter two positive integers: ";
  cin >> m >> n;
  if (m < n) { int temp = m; m = n; n = temp; } // 使 m>=n
  cout << "The g.c.d. of " << m << " and " << n << " is ";
  while (n > 0)
  { r = m % n;
    m = n;
    n = r;
  }
  cout << m << endl;
}
```

```
Enter two positive integers: 532 112
The g.c.d. of 532 and 112 is 28
```

# 第 5 章 函 数

## 5.1 介绍

许多有用的程序比我们想像的要大，为使大程序便于管理，程序员们将它们调整为子程序，这些子程序称为函数。它们可以单独编译和测试，并且可在不同的程序中重用，这种调整是成功的面向对象软件的特征。

## 5.2 标准 C++ 的库函数

标准 C++ 的函数库是预先定义好的函数和其他的程序元素的集合，可以通过头文件来访问。前面已经使用过一些库函数了，如：在 `<climits>` 中定义的 `INT_MAX`（例 2.3），在 `<cmath>` 中定义的 `sqrt()`（例 2.15），在 `<cstdlib>` 中定义的 `rand()`（例 4.26）和在 `<ctime>` 中定义的 `time()`（例 4.28）。下面的第一个例子说明一个数学函数的用法。

### 例 5.1 平方根函数 `sqrt()`

一个给定正数的平方根是平方为给定数的一个数。9 的平方根为 3，因为 3 的平方是 9。可以将平方根函数看做一个“黑匣子”，当输入数字 9，则输出为 3。当输入数字 2 时，则输出为 1.41421。这个函数具有与完整程序相同的输入产生输出的特征。然而，输入产生输出的过程是隐藏的：不需要了解函数是如何由 2 产生出 1.41421 的，所有要了解的就是输出 1.41421 具有平方根的属性：它的平方是输入 2。

下面有一个使用这个预定义的平方根函数的简单程序：

```
#include <cmath>      // 定义 sqrt() 函数
#include <iostream>    // 定义 cout 对象
using namespace std;
int main ()
{ // 测试 sqrt ()
  for (int x=0; x < 6; x++)
    cout << "x\t" << x << "\t" << sqrt(x) << endl;
```

0	0
1	1
2	1.41421
3	1.73205
4	2
5	2.23607

本例打印出数字 0 到 5 的平方根。每次在 for 循环中计算表达式  $\text{sqrt}()$ ， $\text{sqrt}()$  函数就被执行。它的真正代码隐藏在标准 C++ 函数库中。在使用时，可以确定地认为不论此时  $x$  的值是多少，表达式  $\text{sqrt}()$  都将被真正的平方根所代替。

注意在程序第一行的提示：`#include <cmath>`，这是编译器寻找  $\text{sqrt}()$  函数所必需的，它告诉编译器这个函数是在头文件 `<cmath>` 中定义的。

像  $\text{sqrt}()$  一样的函数是通过将它的名字用做语句中的一个变量来执行的，如下所示：

```
y = sqrt(x);
```

这称为函数调用。在例 5.1 中，代码  $\text{sqrt}()$  调用  $\text{sqrt}()$  函数，在圆括号中的表达式  $x$  称为这次函数调用的参数或实参，并且称它是通过值传递给该函数的。因此，当  $x$  为 3 时，值 3 通过调用  $\text{sqrt}(x)$  被传递给  $\text{sqrt}()$  函数。

上述过程在图 5.1 中表示。变量  $x$  和  $y$  在 `main()` 中声明， $x$  的值被传递给  $\text{sqrt}()$  函数，然后，该函数将值 1.73205 返回给 `main()`。注意代表  $\text{sqrt}()$  函数的盒子是用阴影来表示的，这指明它内部的工作机制是不可见的。

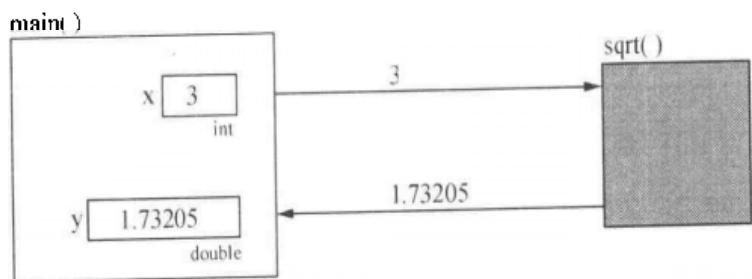


图 5.1  $\text{sqrt}()$  函数的调用过程

### 例 5.2 测试三角式的恒等性

下面是另一个使用头文件 `<cmath>` 的例子，它的目的是用经验证明恒等式  $\sin 2x = 2 \sin x \cos x$ 。

```
int main()
{ // 测试恒等式  $\sin 2x = 2 \sin x \cos x$ 
  for (float x = 0; x < 2; x += 0.2)
    cout << x << "\t\t" << sin(2 * x) << "\t"
          << 2 * sin(x) * cos(x) << "\t\n";
}
```

0	0	0
0.2	0.389418	0.389418
0.4	0.717356	0.717356
0.6	0.932039	0.932039
0.8	0.999574	0.999574
1	0.909297	0.909297
1.2	0.675463	0.675463
1.4	0.334988	0.334988
1.6	-0.0583744	-0.0583744
1.8	-0.442521	-0.442521



本例的运行结果是在第一列打印  $x$ ，在第二列打印  $\sin 2x$ ，在第三列打印  $2 \sin x \cos x$ 。对每个被测试的  $x$  值， $\sin 2x = 2 \sin x \cos x$ 。当然，这并不能证明恒等性，但是它提供了证明其正确性的令人信服的经验性证据。

在表达式中，函数值可以用做一个普通变量。可以这样书写：

```
z = sqrt(2);
cout << 2 * sin(x) * cos(x);
```

甚至可以像下面这样嵌套地调用函数：

```
y = sqrt(1 + 2 * sqrt(3 + 4 * sqrt(5)));
```

许多可以在袖珍计算器上找到的数学函数是在头文件 `<cmath>` 中声明的，如表 5.1 所示：

表 5.1 一些在头文件 `<cmath>` 中定义的函数

Function	描述	举例
<code>acos(x)</code>	$x$ 的余弦	<code>acos(0.2)</code> 返回 1.36944
<code>asin(x)</code>	$x$ 的反正弦	<code>asin(0.2)</code> 返回 0.201358
<code>atan(x)</code>	$x$ 的反正切	<code>atan(0.2)</code> 返回 0.197396
<code>ceil(x)</code>	$x$ 的 ceiling	<code>ceil(3.141593)</code> 返回 4.0
<code>cos(x)</code>	$x$ 的余弦	<code>cos(2)</code> 返回 -0.416147
<code>exp(x)</code>	$x$ 的指数	<code>exp(2)</code> 返回 7.38906
<code>fabs(x)</code>	$x$ 的绝对值	<code>fabs(-2)</code> 返回 2.0
<code>floor(x)</code>	$x$ 的基数	<code>floor(3.141593)</code> 返回 3.0
<code>log(x)</code>	$x$ 的自然对数	<code>log(2)</code> 返回 0.693147
<code>log10(x)</code>	$x$ 的常用对数	<code>log10(2)</code> 返回 0.30103
<code>pow(x, p)</code>	$x$ 的 $p$ 次幂	<code>pow(2, 3)</code> 返回 8.0
<code>sin(x)</code>	$x$ 的正弦	<code>sin(2)</code> 返回 0.909297
<code>sqr(x)</code>	$x$ 的平方根	<code>sqr(2)</code> 返回 1.41421
<code>tan(x)</code>	$x$ 的正切	<code>tan(2)</code> 返回 -2.18504

注意，每个数学函数返回的是一个 `double` 类型。如果将一个整数传递给函数，在函数处理之前该整数会转换成一个 `double` 类型的数。

表 5.2 列出了标准的 C++ 函数库中一些非常有用的头文件。

表 5.2 在标准的 C++ 库中的一些头文件

文件	描述
<code>&lt;cassert&gt;</code>	定义 <code>assert()</code> 函数
<code>&lt;ctype&gt;</code>	定义测试字符的函数
<code>&lt;efloat&gt;</code>	定义与浮点数相关的常量
<code>&lt;climit&gt;</code>	定义在你的本地系统上的整数限制
<code>&lt;cmath&gt;</code>	定义数学函数

(续表)

文件	描述
<stdio>	定义标准的输入和输出函数
<stdlib>	定义公用函数
<string>	定义处理字符串的函数
<time>	定义时间和日期函数

这些头文件源自标准 C 函数库, 它们的用法与 <iostream> 等标准 C++ 头文件的用法相同。例如, 如果想使用头文件 <stdlib> 中的随机数函数 rand(), 则在主程序的开始应包含下面的预处理伪指令:

```
# include <stdlib>
```

有关标准 C 函数库的更详尽的介绍参见第八章和附录 F。

## 5.3 用户自定义函数

标准的 C++ 函数库所提供的多种多样的函数仍然不能满足更多的程序需要, 程序员还需要可以定义自己的函数。

### 例 5.3 cube() 函数

下面是一个简单的用户自定义函数的例子:

```
int cube ( int x)
{ // 返回 x 的立方
    return x * x * x;
}
```

该函数返回传递给它的整数的立方。因此调用 cube(2) 将返回 8。

一个用户自定义函数由两部分组成: 函数头和函数体。函数头的语法为:

返回值类型 函数名 (参数表)

其中, 为编译器指定了该函数的返回值类型、函数的名字和它的参数列表。在例 5.3 中, 函数的返回值类型为 int, 函数名为 cube, 参数表为 int x。所以它的函数头为:

```
int cube (int x)
```

函数体是跟在函数头后面的语句块, 包含执行函数动作的代码, 其中所包含的 return 语句指定该函数返回到调用处的值。cube 函数的函数体为:

```
{ // 返回 x 的立方
    return x * x * x;
}
```

这大概是最简单的函数体了。通常的函数体更大，但是典型的函数头是写在一行代码中的

注意 `main()` 本身就是一个函数，它的函数头为：

```
int main ()
```

它的函数体是程序本身，返回值类型是 `int`，函数名是 `main`，参数表为空。

函数的返回语句有两个作用：终止函数的执行；并且返回一个值给调用程序。它的语法为：

```
return 表达式；
```

其中的表达式是与函数返回值的类型相同的任意变量的表达式。

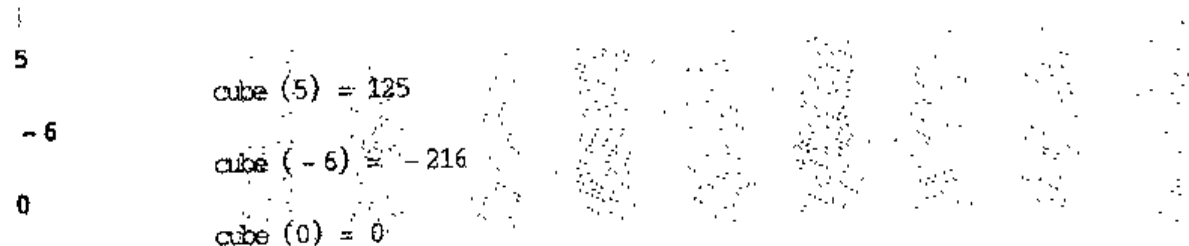
## 5.4 测试程序

在创建自己的函数时，应该立即用一个简单的程序测试该函数，这样的程序称为测试程序，它的目的就是测试函数。它是一个简单的、临时的、特别的程序。这句话的意思是测试程序中不需要包括所有的通常很精确的东西，如用户提示、输出书签和文件，一旦用它测试完创建的函数后，就可以抛弃它。

### 例 5.4 `cube()` 函数的测试程序

下面是一个包含有例 5.4 中 `cube()` 函数的定义和它的测试程序的完整程序：

```
int cube (int x);  
// 返回 x 的立方  
return x * x * x;  
  
int main ()  
{ // 测试 cube () 函数  
    int n=1;  
    while (n != 0)  
    { cin >> n;  
      cout << "\ncube (" << n << ") = " << cube (n) << endl;  
    }  
}
```



本例读入整数并打印它们的立方，直到用户输入标记 0。每个读入的整数通过调用 `cube(n)` 传递给 `cube()` 函数。函数的返回值代替表达式 `cube(n)`，然后传递给输出 `cout`。

在图 5.2 中, 可以看到 main () 函数与 cube () 函数之间的联系:

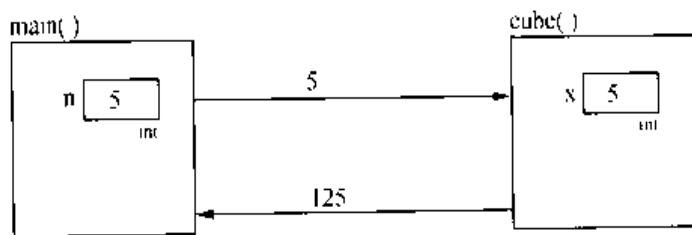


图 5.2 cube 函数的调用过程

main () 函数传递值 5 给 cube () 函数, 然后 cube () 函数返回值 125 给 main () 函数。参数 n 通过形参 x 进行值传递, 图 5.2 中简单地表示出当调用函数时, x 被赋值为 n。

下面的例子说明一个名为 max () 的用户自定义函数, 返回传递给它的两个 int 型数的最大值, 该函数有两个参数。

### 例 5.5 max () 的测试程序

下例是一个带两个参数的函数, 它返回传递来的两个值中较大的一个:

```
int max (int x, int y)
| // 返回两个给定整数中的最大数
|   if (x < y) return y;
|   else return x;
|
int main ()
| // 测试 max () 函数
|   int m, n;
|   do
|   { cin >> m >> n;
|     cout << " \tmax(" << m << ", " << n << ") = " << max(m,n) << endl;
|
|     while (m != 0);
```

```
5 8      max (5, 8) = 8
4 -3     max (4, -3) = 4
0 0      max (0, 0) = 0
```

注意该函数中有不止一个返回语句。其中第一个返回语句用来终止函数的执行并返回确切的值给调用程序。

return 语句类似于 break 语句, 它是一条跳出包含它的函数的跳转语句。虽然通常情况下 return 语句是放在函数的最后, 它也可以放在函数内部任意语句可以出现的地方。

## 5.5 函数声明和定义

上面的两个例子说明了在一个程序中定义函数的一种方法：在主程序的上面列出一个函数的完整定义。这是一种最简单的方法，并且有助于使用测试程序。

另一种更常见的方法是在主程序的上面仅列出函数头，然后将函数的完整定义（函数头和函数体）放在主程序的下面。在下面的例子中说明了这种方法。

在这种方法中，函数的声明与定义是分开的。函数的声明仅是简单的函数头，后跟一个分号。函数的定义是完整的函数：包括函数头和函数体。函数的声明又称为函数原型。

函数的声明如同一个变量声明：它的目的只是提供给编译器所需要的所有信息，以便编译器对后面的文件进行编译。编译器不需要知道函数是如何工作的（函数体），它仅仅需要知道函数名、函数参数的个数和类型、及函数的返回值，这些信息都包含在函数头中。

像变量声明一样，函数声明必须在函数名被使用之前出现。但是函数的定义是与函数声明分开的，可以在 `main()` 函数以外的任意位置出现，它通常是列在 `main()` 函数之后，或者作为一个单独的文件。

在函数的参数表中列出的变量称为形参，它们是仅在函数执行时存在的局部变量。在形参列表中包含了形参的声明，在上例中，形参是 `x` 和 `y`。

在函数调用中列出的变量称为实参，像主程序中其他的变量一样，它们必须在调用前声明。在上例中，实参是 `m` 和 `n`。

在这些例子中，实参是通过值传递的，这表示它们的值被赋给函数中相应的形参，所以在上面的例子中，`m` 的值被赋给 `x`，`n` 的值被赋给 `y`。当通过值传递时，实参可以为常量或一般的表达式。例如，`max()` 函数可以通过 `max(44, 5 * m - n)` 调用，将会把 44 赋给 `x`，而将表达式 `5 * m - n` 的值赋给 `y`。

### 例 5.6 声明与定义分开的 `max()` 函数

本例是与例 5.6 中相同的 `max()` 函数的测试程序。但是这里函数的声明出现在主程序的上面，而函数的定义跟在主程序之后。

```
int max(int, int);  
// 返回两个给定整数中的最大数  
  
int main()  
{ // 测试 max() 函数  
    int m, n;  
    do  
    { cin >> m >> n;  
      cout << " \tmax(" << m << ", " << n << ") = " << max(m,n) << endl;  
    }  
    while (m != 0);  
}
```

```
int max (int x, int y)
{ if (x < y) return y;
  else return x;
}
```

注意形参  $x$  和  $y$  是出现在函数定义的函数头之中（像通常一样），而不是出现在函数声明中。

函数声明与变量声明非常相似，特别是函数没有参数时。例如，在一个处理字符串的程序中，可能需要一个名为 `length` 的变量来存放字符串的长度，但是可以合理地改变为使用一个函数，在需要的时候计算字符串的长度，而不是存储和修改该值。这个函数可以声明为：

```
int length ();
```

而变量可以声明为：

```
int length;
```

两者看起来惟一的区别在于函数的声明包含一个圆括号（）实际上，这两种方法是很不相同的，但它们使用时在语句构成上非常类似。这时，可以把函数看做一个“活动的变量”，即一个可以工作的变量。

### 例 5.7 单独编译

函数定义经常在单独的文件中独立编译。例如，在标准 C++ 函数库中的所有函数都是单独编译的。这样做的一个原因是“信息隐藏”——即程序完全编译所需的信息在程序员理解程序时不是必需的，这些信息就被隐藏起来。经验表明信息隐藏有利于理解程序，并且对成功地开发大的软件项目很有帮助。

### 例 5.8 单独编译的 `max()` 函数

本例说明了将 `max` 函数和它的测试程序分开编译的一种方法。测试程序是一个名为 `test_max.cpp` 的文件，而函数是一个名为 `max.cpp` 的文件。

`test_max.cpp`

```
int max (int, int);
// 返回两个给定整数中的最大数

int main ()
{ // 测试 max () 函数
  int m, n;
  do
  { cin >> m >> n;
    cout << " \tmax(" << m << ", " << n << ") = " << max(m,n) << endl;
  }
  while (m != 0);
}
```

```
max.cpp
int max (int x, int y)
{ if (x < y) return y;
  else return x;
}
```

实际上，可以用来一起编译这些文件的命令依赖于所使用的系统。在 UNIX 中，就可以这样做：

```
$c++ -o max.o max.c
$c++ -o test_max.o test_max.c max.o
$test_max
```

(其中，\$符号是系统提示信息)。第一个命令编译 max 函数，第二个命令单独编译测试程序，第三个命令将它们链接起来以生成可执行的模块 test\_max，然后第四个命令运行 test\_max。

单独编译的好处之一是在书写调用程序之前可以对函数进行单独测试。一旦知道 max 函数可以正常工作，就可以忘掉它是如何工作的，并且可以将它存为一个“黑匣子”，以备在任意需要的地方使用。在数学函数库中的函数就是这样来使用的，这是“现成软件”的观点。

单独编译的另一个好处是可以很容易地用一个模块代替另一个与它等价的模块。例如，如果碰巧发现了一个更好的方法，可以计算两个整数的最大值，则可以编译并测试这个函数，然后将该模块连接到任意一个使用前面的 max() 函数的程序中。

## 5.6 局部变量和函数

一个局部变量是在一个语句块的内部声明的简单变量，它仅可以在该语句块的内部被访问。由于函数体本身就是一个语句块，在函数内部声明的变量就是该函数的局部变量。它们仅在函数执行时存在。函数的形参也可以看做函数的局部变量。

下面的两个例子展示了带有局部变量的函数。

### 例 5.9 阶乘函数

在例 4.9 中介绍了阶乘函数。一个正整数  $n$  的阶乘是数字  $n!$ ，其中包含小于  $n$  的所有正整数  $n$  的乘积。

$$n! = (n)(n-1) \cdots (3)(2)(1)$$

例如： $5! = (5)(4)(3)(2)(1) = 120$

下面是阶乘函数的一种实现方法：

```
long fact (int n)
{ // 返回  $n! = n * (n-1) * (n-2) * \dots * (2) * (1)$ 
```

```

    if (n < 0) return 0;
    int t = 1;
    while (n > 1)
        t *= n--;
    return t;
}

```

这个函数有两个局部变量  $n$  和  $t$ 。参数  $n$  是局部变量，因为它是在函数的参数表中声明的。变量  $t$  是局部变量，因为它是在函数体的内部声明的。

下面是这个阶乘函数的测试程序：

```

long fact (int)
// 返回  $n! = n \times (n-1) \times (n-2) \times \dots \times (2) \times (1)$ 

int main ()
// 测试 factorial () 函数
for (int i = 1; i < 6; i++)
    cout << " " << fact (i);
cout << endl;
}
0 1 1 2 6 24 120

```

这个测试程序能够单独进行编译，或者它也可以和这个阶乘函数放在一个文件中同时编译。

### 例 5.10 排列函数

排列是指取自一个有限集合的元素的一个排列。排列函数  $P(n, k)$  给出从  $n$  个项目中任取  $k$  个项目的不同排列的个数，计算这个函数的一种方法是使用公式：

$$P(n, k) = \frac{n!}{(n-k)!}$$

例如：

$$P(5, 2) = \frac{5!}{(5-2)!} = \frac{5!}{3!} = \frac{120}{6} = 20$$

这是 5 中取 2 的 20 个不同的排列。例如，从集合  $\{A, B, C, D, E\}$  中任取 2 个的 20 种不同的排列为：AB, AC, AD, AE, BC, BD, BE, CD, CE, DE, BA, CA, DA, EA, CB, DB, EB, DC, EC, ED。

以下代码利用这个公式实现排列函数：

```

long perm (int n, int k)
// 返回  $p(n, k)$ ,  $k$  的从  $n$  开始的排列数
if (n < 0 || k < 0 || k > n) return 0;
return fact (n) / fact (n - k);
}

```

注意，其中条件  $(n < 0 || k < 0 || k > n)$  用来处理参数越界的情况，这时，函数返回一个“不可能的值”0，以表明输入有错，该值可以被调用函数作为一个“错误标志”识别。



下面是 perm () 函数的测试程序:

```
long perm (int, int);
// 返回 p (n, k), k 的从 n 开始的排列数

int main ()
{ // 测试 perm () 函数
  for (int i = -1; i < 8; i++)
    { for (int j = -1; j <= i+1; j++)
      { cout << " " << perm (i, j);
        cout << endl;
      }
    }

  cout << endl;

  cout << "0\n";
  cout << "0 1 0\n";
  cout << "0 1 1 0\n";
  cout << "0 1 2 2 0\n";
  cout << "0 1 3 6 6 0\n";
  cout << "0 1 4 12 24 24 0\n";
  cout << "0 1 5 20 60 120 120 0\n";
  cout << "0 1 6 30 120 360 720 720 0\n";
  cout << "0 1 7 42 210 840 2520 5040 5040 0\n";
}
```

注意, 该测试程序检查“当  $i < 0$ ,  $j < 0$ , 且  $j > i$  时的特殊情况”, 这样的值被称为临界值, 因为它们位于输出集合的边界 (在那里 perm () 返回 0 值)。

## 5.7 void 函数

void 函数是不需要返回值的函数。在其他的程序设计语言中, 这样的函数被称为过程或例程。在 C++ 中, 这样的函数是通过用 void 作为函数返回值类型来标识的。

类型用来指定一个值的集合。例如, short 类型指定从 -32 768 到 32 767 的整数的集合, void 类型用来指定空集。因此, 变量不能被声明为 void 类型。一个 void 函数是没有返回值的简单函数。

### 例 5.11 打印日期的函数

```
void printDate (int, int, int);
// 按文字的方式打印给定的日期

int main ()
{ // 测试 printDate () 函数
  int month, day, year;
  do
  { cin >> month >> day >> year;
    printDate (month, day, year);
  }
  while (month > 0);
}
```

```

void printDate (int m, int d, int y)
{ // 按文字的方式打印给定的日期
  if (m < 1 || m > 12 || d < 1 || d > 31 || y < 0)
  { cerr << "error: parameter out of range. \n";
    return;
  }
  switch (m)
  { case 1:  cout << "January ";   break;
    case 2:  cout << "February ";  break;
    case 3:  cout << "March ";     break;
    case 4:  cout << "April ";     break;
    case 5:  cout << "May ";       break;
    case 6:  cout << "June ";      break;
    case 7:  cout << "July ";      break;
    case 8:  cout << "August ";    break;
    case 9:  cout << "September "; break;
    case 10: cout << "October ";   break;
    case 11: cout << "November ";  break;
    case 12: cout << "December ";  break;
  }
  cout << d << ", " << y << endl;
}

```

**12 7 1941**

December 7, 1941

**5 16 1994**

May 16, 1994

**0 0 0**

Error: parameter out of range.

printDate() 函数没有返回值，它的目的只是打印日期，所以它的返回值类型是 void。该函数使用了一个 switch 语句，将月份打印成文字，并将日期和年份打印成整数。

注意，如果参数明显越界（例如， $m > 12$  或  $y < 0$ ），则该函数什么都不打印而返回。但是，可能有些值如 February 31, 1996 将会被打印。如何纠正这个错误留作练习。

由于一个 void 函数没有返回值，它不需要包含 return 语句。如果其中包含有 return 语句，那它应该以如下形式出现：

```
return;
```

在关键字 return 后面没有跟表达式。在这种情况下，return 语句的作用只是终止函数。

没有返回值的函数只是完成一个动作，因此，通常最好使用一个动词作为函数名。例如，上面的函数名为 printDate，而不是像 date 一样的名词。

## 5.8 布尔函数

在某些情况下，使用一个函数来作为条件是很有用的，特别是在没有 if 或 while 语句时。在英国的逻辑学家 George Boole (1815 ~ 1864) 创立了布尔代数学之后，这种函数被称

为布尔函数。

### 例 5.12 字符分类

下例将 128 个 ASCII 码字符分类（参见附录 A）：

```
#include <cctype>      // 定义函数 isdigit (), islower () .....等等
#include <iostream>    // 定义 cout 对象
using namespace std;

void printCharCategory (char c);
// 打印给定字符属于的类型

int main ()
{ // 测试 printCharCategory 函数
  for (int c = 0; c < 128; c++)
    printCharCategory (c);
}

void printCharCategory (char c)
{ // 打印给定字符属于的类型
  cout << "The character [" << c << "], is a ";
  if (isdigit (c)) cout << "digit. \n";
  else if (islower (c)) cout << "lower-case letter. \n";
  else if (isupper (c)) cout << "capital letter. \n";
  else if (isspace (c)) cout << "white space character. \n";
  else if (isctrl (c)) cout << "control character. \n";
  else if (ispunct (c)) cout << "punctuation mark. \n";
  else
    cout << "Error. \n";
}
```

void 函数 printCharCategory () 调用六个布尔函数：isdigit (), islower (), isupper (), isspace (), isctrl (), ispunct (), 其中每个函数都是在头文件 <cctype> 中预定义的。这些函数被用来测试目标的字符类型（即“c type”）。

下面是部分输出：

```
The character [ ] is a white space character.
The character [!] is a punctuation mark.
The character ["] is a punctuation mark.
The character [#] is a punctuation mark..
The character [$] is a punctuation mark..
```

整个输出包含 128 行。

本例说明了一些新的思想。主要的思想是六个布尔函数 isdigit (), islower (), isupper (), isspace (), isctrl (), ispunct () 的使用方法。例如，调用 isspace (c) 测试字符 c 是否是一个空白符（有六个空白符：水平制表符 \t、换行符 \n、垂直制表符 \v、换页符 \f、回车符 \r 和空格符）。如果 c 是这些字符中的任意一个，则函数返回一个表示 true 的非 0 整数；否则返回一个表示 false 的 0。将函数调用做为 if 语句中的条件，会引起相应的输出语句执行，当且

仅当  $c$  是其中一个字符。

每个字符在 `printCharCategory()` 函数中被测试。不使用这个单独的函数，程序也可以完成任务，但使用这个函数使程序更加模块化和结构化。可以确认的基本编程原则是：建议每个任务都作为一个单独的函数编写。

在 C 的头文件（例如 `<cctype>`）中定义的函数，如 `isdigit()` 和 `ispunct()`，最初是为 C 语言定义的。由于 C 语言没有标准的布尔类型，这些布尔函数返回一个整数，而不是 `true` 或 `false`。但是，既然这些 C++ 的布尔值是作为整数存储的（参见 2.2 节），从整数值到布尔值的类型转换是自动进行的。

### 例 5.13 测试素数的函数

以下是一个布尔函数，用于判定一个整数是否是素数。

```
bool isPrime (int n)
{ // 如果 n 是素数返回 true，否则返回 false
    float sqrt_n = sqrt (n);
    if (n < 2) return false;           // 0, 1 不是素数
    if (n < 4) return true;            // 2, 3 是最小的素数
    if (n % 2 == 0) return false;      // 2 是惟一的偶数素数
    for (int d=3; d <= sqrt_n; d += 2)
        if (n % d == 0) return false; // n 能被整除
    return true;                       // n 不能被整除
}
```

该函数求一个给定数  $n$  的因子  $d$ ，通过条件  $(n \% d == 0)$  的值测试整除性。当  $d$  是  $n$  的一个因子时该条件为真，于是函数立即返回值 `false`。如果 `for` 循环没找到  $n$  的任意个因子而结束，则函数返回值 `true`。

一旦  $d$  值超过  $n$  的平方根，就可以停止寻找  $n$  的因子，因为如果  $n$  是  $d * a$  的乘积，则其因子中的一个因子一定小于或等于  $n$  的平方根。在循环外定义变量 `sqrt_n`，以使它仅被计算一次。

下面是函数 `isPrime()` 的测试程序和一次测试运行的情况。

```
#include <cmath>      // 定义 sqrt() 函数
#include <iostream>    // 定义 cout 对象
using namespace std;

bool isPrime (int);
// 如果 n 是素数返回 true，否则返回 false

int main ()
{ for (int n=0; n < 80; n++)
    if (isPrime (n)) cout << n << " ";
    cout << endl;
}
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79
```

注意，如前一个例子中使用的“c-type”，可以使用一个动词短语作为函数名。这个函数被命名为 isPrime，使得人们更容易了解函数的用途：代码

```
if (isPrime (n)) . . .
```

几乎和普通的英语短语“if n is prime”相同。

需要注意该函数不是最理想的。在寻找因子时，仅需要检查素数，因为每一个数（非素数）都是素数的惟一乘积。为了修改该函数，使它只检查素数因子，需要存储所有找到的素数，这就需要使用数组了（参见习题 6.22）。

### 例 5.14 一个闰年函数

闰年是在规则日历上增加额外的一天（2 月 29 日）的那一年。闰年的年数是能被 4 整除的，例如，1992 和 1996 是闰年。但是，许多人都不知道这个规则还有例外情况：世纪年不是闰年，例如，1800 和 1900 年就不是闰年。另外，这个例外还有例外的情况：可以被 400 整除的世纪年是闰年，于是，2000 年是闰年。

下面是实现这种定义的布尔函数：

```
bool isLeapYear (int y)
// 当且仅当 y 是闰年时返回 true
return y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
;
```

其中，只有当 y 能被 4 整除但不能被 100 整除，或者 y 也能被 400 整除时，复合条件  $y \% 4 == 0 \ \&\& \ y \% 100 != 0 \ || \ y \% 400 == 0$  才为真。此时，函数返回 true；其他情况返回 false。下面是这个函数的测试程序和测试运行情况：

```
bool isLeapYear (int);
// 当且仅当 y 是闰年时返回 true

int main ()
| // 测试 isLeapYear () 函数
  int n;
  do
  | cin > > n;
  | if (isLeapYear (n)) cout << n << " is a leap year. \n";
  | else cout << n << " is not a leap year. \n";
  |
  | while (n > 1);
|
```

输出为：

```
2000
2000 is a leap year.
2001
2001 is not a leap year.
0
0 is a leap year.
```

## 5.9 I/O 函数

当程序的主要任务与封装该任务所需的大量细节关系不大时，函数就特别有用。例如在处理个人记录时，可能有一个需要以交互方式输入用户年龄的程序。通过把这个任务归为单独的函数，可以把所需的细节封装起来，这样就可以不转移主程序而保证正确的数据入口。

前面已经看过输出函数的例子。在例 5.11 中，函数 `printDate()` 的唯一目的就是打印它的输入参数所代表的数据。该函数把信息传递给标准的输出设备（即计算机的屏幕），而不是将信息回传给调用函数。输入函数与上面的描述类似，它从标准输入设备（即键盘）中读入信息，而不是通过参数得到信息。

下面的例子展示了一个输入函数，在这个例子中，循环控制 `while(true)` 使得循环看起来像一个无限循环；条件 (`true`) 总是为真的。但是循环实际上是由 `return` 语句来控制的，该 `return` 语句不但终止循环，而且终止函数。

### 例 5.15 读入用户年龄的函数

下面的例子是一个简单的函数，提示用户输入他的年龄，然后返回该年龄。由不允许输入任意的无效整数这点而言，该程序是强壮的，函数反复地要求用户输入，直到获得一个 0 到 120 范围内的整数。

```
int age ()
{ // 提示用户输入 (他) 她的年龄，并返回那个值
  int n;
  while (true)
  { cout << "How old are you: ";
    cin >> n;
    if (n < 0) cout << "\a\tYour age could not be negative.";
    else if (n > 120) cout << "\a\tYou could not be over 120.";
    else return n;
    cout << "\n\tTry again. \n";
  }
}
```

一旦从 `cin` 中获得的输入被接受，该函数就由一个 `return` 语句控制，将输入回传给调用函数。如果输入不被接受 ( $n < 0$  或者  $n > 100$ )，则系统通过打印字符 “\a” 发出警告音并打印出一个注释，然后用户被要求“再试一次”。

注意：这是一个没有将 `return` 用在函数结束处的例子。

下面是该函数的测试程序和一次运行的结果。

```
int age ();
// 提示用户输入 (他) 她的年龄，并返回那个值

int main ()
{ // 测试 age () 函数
```

```

int a = age ();
cout << " \nYou are " << a << " years old. \n";
}

How old are you: 125
    You could not be over 120.
    Try again.
How old are you: -3
    You age could not be negative.
    Try again.
How old are you: 99

You are 99 years old.

```

注意函数的参数表为空。但即使没有输入参数，在函数头和每次函数调用时必须要有圆括号 ( )。

## 5.10 引用传递

到目前为止，在函数中看到的参数都是通过值传递的，这意味着在函数开始执行之前，用于函数调用的表达式首先被计算，然后计算结果被赋值给函数参数表中相应的形参。例如，在调用 `cube(x)` 时，如果 `x` 的值为 4，则在函数开始执行语句前，值 4 被传递给局部变量 `n`。由于值 4 仅在函数内部使用，变量 `x` 不受函数影响，因此该变量 `x` 是一个只读的形参。

在函数调用时，值传递机制允许用更一般的表达式来代替实参。例如 `cube()` 函数可以用 `cube(3)` 形式调用，或用 `cube(2 * x - 3)` 形式调用，甚至可以用 `cube(2 * sqrt(x) - cube(3))` 形式来调用。此时圆括号中的表达式被计算成一个值，然后传递给一个函数。

在普通情况下使用函数时，这种通讯的只读、值传递的方法是常用的。这使得函数更加独立，并防止它受到偶然因素的影响。然而，在某些情况下，函数需要改变传递给它的实参的值，这时就需要用到引用传递。

通过引用而不是通过值来传递一个参数，只需要在函数参数表的类型标志符前加一个“&”符号，这使得局部变量成为传递给它的实参的一个引用，所以该实参是可读写的，而不是只读的。并且在函数内部该局部变量的任意改变都会引起传递给它的实参的同样变化。

注意，通过值传递的形参称为值形参，而通过引用传递的形参称为引用形参。

### 例 5.16 swap() 函数

这个小函数可广泛地用于数据排序：

```

void swap (float& x, float& y)
{ // 交换 x, y 的值
    float temp = x;
    x = y;
    y = temp;
}

```

它的作用是交换传递给它的两个数。这是通过将前面的形参  $x$  和  $y$  声明为引用变量实现的： $\text{float\& } x, \text{float\& } y$ 。引用运算符  $\&$  使得  $x$  和  $y$  等同于传递给函数的实参。下面是该函数的测试程序和运行结果。

```
void swap (float&, float&);
// 交换 x, y 的值
```

```
int main ()
{ // 测试 swap () 函数
  float a = 22.2, b = 44.4;
  cout << "a = " << a << ", b = " << b << endl;
  swap (a, b);
  cout << "a = " << a << ", b = " << b << endl;
```

$a = 22.2, b = 44.4$

$a = 44.4, b = 22.2$

当调用  $\text{swap}(a, b)$  执行时，函数创建它的局部引用  $x$  和  $y$ ，因此， $x$  是  $a$  在函数内部的名字，而  $y$  是  $b$  在函数内部的名字。然后，函数执行 3 个语句：声明局部变量  $\text{temp}$  并将  $\text{temp}$  初始化为  $x$  的值（即  $a$  的值）； $x$ （即  $a$ ）被赋值为  $y$ （即  $b$ ）； $y$ （即  $b$ ）被赋值为  $\text{temp}$  的值。所以，最后  $a$  的值为  $44.4$ ，而  $b$  最后的值为  $22.2$ ，如图 5.3 所示。

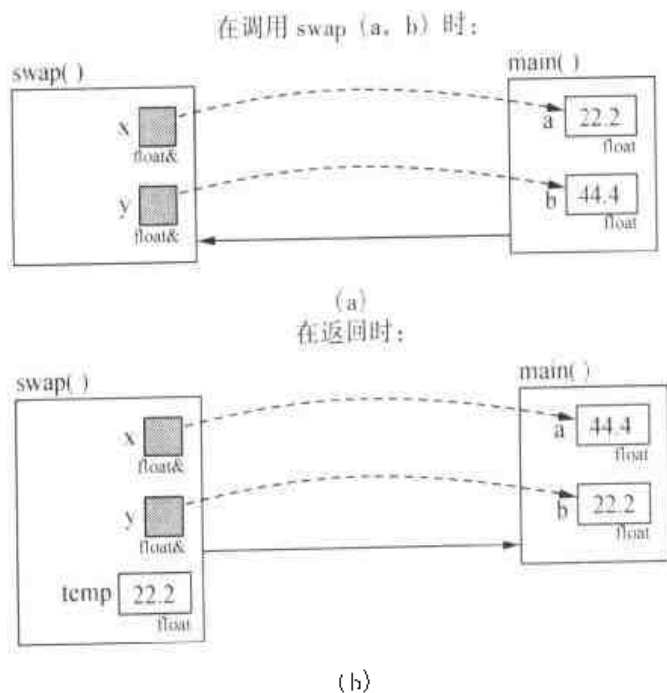


图 5.3  $\text{swap}$  的调用过程

注意函数的声明

```
void swap (float& x, float& y);
```

其中每个引用形参都包含了引用运算符  $\&$  符号，即使该形参被删除。

一些程序中引用运算符是作为形参的前缀而不是作为类型的后缀书写的，如下所示：

```
void swap (float&x, float&y)
```



对于 C 程序员来说这种风格更常见。在 C++ 中，把 `x` 看做是一个形参，而 `float&` 看做是它的类型。但是，编译器将接受 `float& x`，`float &x`，`float & x`，甚至 `float& x`，这是习惯问题。

### 例 5.17 值传递和引用传递

本例说明值传递和引用传递的区别：

```
void f (int, int&);
// 改变引用函数为 99

int main ()
{ // 测试 f () 函数
  int a = 22, b = 44;
  cout << "a = " << a << ", b = " << b << endl;
  f (a, b);
  cout << "a = " << a << ", b = " << b << endl;
  f (2*a-3, b);
  cout << "a = " << a << ", b = " << b << endl;
}

void f (int x, int& y)
{ // 改变引用函数为 99
  x = 88;
  y = 99;
}

a = 22, b = 44
a = 22, b = 99
a = 22, b = 99
```

`f (a, b)` 调用通过值传递将 `a` 传递给 `x`，通过引用传递把 `b` 传递给 `y`。所以 `x` 是一个局部变量，被赋值为 `a` 的值 22，而 `y` 是值为 33 的变量 `b` 的一个别名。函数将 88 赋给 `x`，但不影响 `a`。但是当函数将 99 赋给 `y` 时，它确实将 99 赋给了 `b`，因为 `y` 是 `b` 的别名。因此，当函数终止时，`a` 的值仍为其初值 22，而 `b` 有了一个新的值 99。实参 `a` 是只读的，而实参 `b` 是可读写的，如图 5.4 所示。

表 5.3 总结了值传递和引用传递的区别。

表 5.3 值传递和引用传递

值传递	引用传递
<pre>int x;</pre> 参数 <code>x</code> 是一个局部变量 它是实参的副本 它不能改变实参 通过值传递的参数可以是一个常量、一个变量或一个表达式 实参是只读的	<pre>int &amp;x;</pre> 参数 <code>x</code> 是一个局部引用 它是实参的同义字 它能改变实参 通过引用的参数必须为一个变量 实参是可读写的

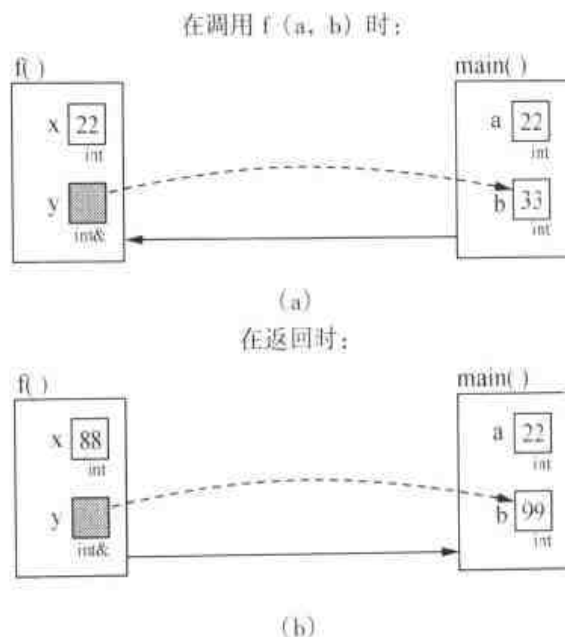


图 5.4 f 的调用过程

通常需要使用引用形参的情况是，函数必须返回多于一个的值。函数只能通过 return 语句直接返回一个值，所以如果需要返回多于一个的值，就需要用到引用形参。

### 例 5.18 返回多于一个的值

本函数通过使用两个引用形参 area 和 circumference 返回两个值，分别是给定半径的圆的面积和周长。

```
void computeCircle (double& area, double& circumference, double r)
{ // 返回半径为 r 的一个圆的面积和周长
    const double PI = 3.141592653589793;
    area = PI * r * r;
    circumference = 2 * PI * r;
}
```

下面是该函数的测试程序和一次运行结果：

```
void computeCircle (double&, double&, double);
// 返回半径为 r 的一个圆的面积和周长

int main ()
{ // 测试 computeCircle () 函数
    double r, a, c;
    cout << "Enter radius: ";
    cin >> r;
    computeCircle (a, c, r);
    cout << "area = " << a << ", circumference = " << c << endl;
}

Enter radius: 100
area = 31415.9, circumference = 628.319
```

注意，输出参数 area 和 circumference 是在参数列表的开始列出的，在输入参数 r 的左

边,这种标准的C风格是与赋值语句的格式:  $y = x$  相一致的,其中值的信息从右面的只读变量  $x$  传给左边的可读写变量  $y$ 。

## 5.11 通过常量引用传递

通过引用传递形参有两个理由,其一:如果函数必须改变实参的值(如函数 `swap()` 所做的),则必须使用引用传递;另外,如果被传递给一个函数的实参要占用大量的存储空间(例如,一个1MB的图像),则使用引用传递将更加有效,这样可以避免复制参数。然而,这种方法也允许函数改变实参的值(即内容)。如果不希望函数修改它的内容(例如,如果函数的目的是打印对象),则通过引用传递是不安全的。所幸的是,C++提供了第三种选择:通过常量引用传递。它的工作原理与引用传递基本一致,不同的是禁止函数修改参数的值。结果是,函数可以通过它的形参的别名来存取实参,但是在函数的执行过程中不允许改变形参的值。通过值传递的参数被称为“只读的”,因为不能写(即修改)参数的内容。

### 例 5.19 通过常量引用传递

本例说明了三种将参数传递给函数的方法:

```
void f(int x, int& y, const int& z)
{
    x += z;
    y += z;
    cout << "x = " << x << ", y = " << y << ", z = " << z << endl;
}
```

第一个参数  $a$  是通过值传递的,第二个参数  $b$  是通过引用传递的,而第三个参数  $c$  是通过常量引用传递的。

```
void f(int, int&, const int&);
int main()
{
    // 测试 f() 函数
    int a = 22, b = 33, c = 44;
    cout << "a = " << a << ", b = " << b << ", c = " << c << endl;
    f(a, b, c);
    cout << "a = " << a << ", b = " << b << ", c = " << c << endl;
    f(2*a+3, b, c);
    cout << "a = " << a << ", b = " << b << ", c = " << c << endl;
}

a = 22, b = 33, c = 44
x = 66, y = 77, z = 44
a = 22, b = 77, c = 44
x = 85, y = 121, z = 44
a = 22, b = 121, c = 44
```

函数修改了形参  $x$  和  $y$ ,但不能修改  $z$ 。 $x$  值的改变对实参  $a$  没有影响,因为它通过值传递的。 $y$  值的改变使实参  $b$  发生同样的改变,因为它通过引用传递的。

通过常量引用传递参数通常用于处理大对象的程序中，比如在后面章节中介绍的数组和类实例。基本数据类型的对象（integer, floats 等）通常使用值传递（如果不希望函数改变它们的值）或引用传递（如果希望函数改变它们的值）。

## 5.12 内联函数

函数调用的代价很高，必须使用额外的时间和空间来调用函数，将参数传递给它，为其局部变量分配存储单元，存储变量的当前值和在主程序中执行的位置等。某些时候，最好通过指定函数内联来避免以上这些情况，这会告诉编译器直接用函数的代码来替换每次的函数调用。对程序员而言，除了要使用内联说明符之外，内联函数与普通的函数没有什么不同。

### 例 5.20 内联的 cube 函数

本例与例 5.3 的结果相同。

```
inline int cube (int x)
{ // 返回 x 的立方:
    return x*x*x;
}
```

本例与例 5.3 唯一的区别是，使用了关键字 inline 作为函数头的前缀，这样可以告诉编译器用真正的代码  $(n) \times (n) \times (n)$  代替主程序中的表达式 `cube(n)`。于是测试程序

```
int main ()
{ // 测试 cube () 函数
    cout << cube(4) << endl;
    int x, y;
    cin >> x;
    y = cube(2*x-3);
}
```

将被编译为下面这样的程序：

```
int main ()
{ // 测试 cube () 函数
    cout << (4) * (4) * (4) << endl;
    int x, y;
    cin >> x;
    y = (2*x-3) * (2*x-3) * (2*x-3);
}
```

当编译器用函数真正的代码代替内联函数调用时，称它扩展了内联函数。

C++ 标准并不是真正地要求编译器扩展内联函数，它只是“建议”编译器这样做。不听从这个“建议”的编译器仍然可以被认为是标准的 C++ 编译器。另一方面，一些标准的 C++ 编译器可以扩展某些简单的函数，即使这些函数并没有被声明为内联函数。

警告：使用内联函数会产生负面影响。例如，内联一个在 26 个不同的地方被调用的 40

行的函数，会在程序中至少增加 1000 行不被注意的源代码。内联函数还会限制代码的跨平台性。

## 5.13 作用域

在 3.5 节中介绍了变量名的作用域。一个变量名的作用域是可以使用该变量名的那部分程序。它是从该变量名的声明开始的。如果声明是在函数（包括 `main()` 函数）内部，则作用域延伸到包含有该声明的最内层语句块的最后。

一个程序可以有几个同名的不同对象，前提是这些对象的作用域是嵌套或不相邻的。下面的例子说明了这一点，其运行结果与例 3.7 相同。

### 例 5.21 嵌套和平行的作用域

本例中，`f()` 和 `g()` 是全局函数，第一个 `x` 是全局变量。所以它们的作用域包括整个文件。这称为文件范围。第二个 `x` 是在 `main()` 函数内部声明的，所以它的作用域是局部的，即只有在 `main()` 函数内部才可以访问。第三个 `x` 是在一个内部的语句块中声明的，所以它的作用域就局限在该语句块的内部。每个 `x` 的作用域都重载了前面声明的 `x` 的作用域，所以，当标识符 `x` 被引用时不会产生模糊性。作用域确定运算符 `::` 是用来存取作用域被重载的 `x` 的；此时，全局变量 `x` 的值为 11。

```
void f();    // f() 是全局的
void g();    // g() 是全局的
int x = 11;  // 这个 x 是全局的

int main()
{ int x = 22;
  { int x = 33;
    cout << "In block inside main(): x = " << x << endl;
    // 内部模块范围的结束
  }
  cout << "In main(): x = " << x << endl;
  cout << "In main(): :: x = " << :: x << endl;    // 访问全局 x
  f();
  g();
} // main() 范围结束

void f()
{ int x = 44;
  cout << "In f(): x = " << x << endl;
} // f() 范围结束

void g()
{ cout << "In g(): x = " << x << endl;
} // g() 范围结束
```

```

In block inside main(): x = 33
In main(): x = 22
In main()::: x = 11
In f(): x = 44
In g(): x = 11

```

初值为 44 的  $x$  的作用域限于函数  $f()$  内部，是与  $main()$  平行的；但是它的作用域同时嵌套在全局变量第一个  $x$  的作用域中，所以它的作用域重载了在  $f()$  内部的第一个  $x$ 。在本例中，第一个  $x$  的作用域没有被重载的地方只有  $g()$  函数内部。

## 5.14 重载

C++ 允许不同的函数使用相同的名字。只要它们有不同的参数类型表，编译器就会将它们看做是不同的函数。为区别起见，参数表中必须包含不同个数的参数，或者至少参数类型不同，或参数排列顺序不同。

### 例 5.22 max () 函数的重载

例 5.6 为两个整数定义了函数  $\max()$ 。下面在同一个程序中定义另外两个不同的  $\max()$  函数：

```

int max (int, int);
int max (int, int, int);

int main ()
{ cout << max (99, 77) << " " << max (55, 66, 33);
  }

int max (int x, int y)
| // 返回两个给定数的最大值
  return (x > y ? x : y);
|

int max (int x, int y, int z)
| // 返回两个给定函数的最大值
  int m = (x > y ? x : y); // m = max (x, y)
  return (z > m ? z : m);
|
|
99 66

```

这里定义了三个不同的函数，名字都叫  $\max$ 。编译器检查它们的参数列表以决定在每次调用时使用哪个函数。例如，第一次调用传递两个  $\text{int}$ ，所以有两个  $\text{int}$  参数的函数被调用（如果该函数被删除，则系统将两个整数 99 和 77 转换为双精度型的 99.0 和 77.0，然后将它们传递给有两个双精度型参数的函数）。

函数的重载在 C++ 中被广泛使用，在第 12 章类的使用中，它的作用更加明显。

## 5.15 main () 函数

每个C++程序都需要有一个名为main ()的函数。事实上,可以将整个程序看做是由main ()函数和其中直接或间接调用的其他函数所组成的。程序从调用main ()函数开始执行。

由于main ()函数是一个返回值类型为int型的函数,它通常是这样结束的:

```
return 0;
```

即使许多编译器并不要求这样做。某些编译器允许删去这个语句,但是将会产生一个警告错。返回给操作系统的整数值可以是错误的个数,默认值为0。

在main ()中的return语句可以用来终止非正常的程序,如下例所示。

### 例 5.23 使用 return 语句终止程序

```
int main ()
| // 打印两个输入整数的商
|   int n, d;
|   cout << "Enter two integer: ";
|   cin >> n >> d;
|   if (d == 0) return 0;
|   cout << n << "/" << d << " = " << n/d << endl;
```

```
Enter two integers: 99 17
99/17 = 5
```

如果用户输入d的值为0,则程序将会在没有输出的情况下终止。

```
Enter two integers: 99 0
```

在任何函数中,return语句将会终止当前函数并返回调用函数,这就是在main ()中的return语句能终止程序的原因。有四种方法可以非正常地(即在程序执行到达主语句块的最后之前)终止一个程序。

- (1) 在main ()中使用return语句;
- (2) 调用exit ()函数;
- (3) 调用abort ()函数;
- (4) 抛出一个非外因引起的异常。

exit ()和abort ()函数的介绍见附录F。

exit ()函数是在头文件<cstdlib>中定义的。它的作用体现在终止一个不包含在main ()中的程序。下面的例子说明了这一点。

### 例 5.24 使用 exit () 函数终止程序

```
#include <cstdlib> // 定义 exit () 函数
#include <iostream> // 定义 cin 和 cout 对象
```

```
using namespace std;
double reciprocal (double x);

int main ()
{ double x;
  cin >> x;
  cout << reciprocal (x);
}

double reciprocal (double x)
{
  if (x == 0) exit (1); // 中止程序
  return 1.0/x;
}
```

如果用户输入 0 给  $x$ ，程序将从函数 `reciprocal()` 中终止，而不使用  $x$  作除数。

## 5.16 默认的参数

在 C++ 中，大量的函数参数可以在运行时间内变化。这是通过给可选的参数提供默认值来实现的。

### 例 5.25 默认参数

这个函数计算 3 次多项式  $a_0 + a_1x + a_2x^2 + a_3x^3$ 。实际是使用 horner 算法来进行计算的，通过计算  $a_0 + (a_1 + (a_2 + a_3x)x)x$  来提高效率：

```
double p (double, double, double=0, double=0, double=0);

int main ()
{ // 测试 p () 函数
  double x = 2.0003;
  cout << "p (x, 7) = " << p (x, 7) << endl;
  cout << "p (x, 7, 6) = " << p (x, 7, 6) << endl;
  cout << "p (x, 7, 6, 5) = " << p (x, 7, 6, 5) << endl;
  cout << "p (x, 7, 6, 5, 4) = " << p (x, 7, 6, 5, 4) << endl;
}

double p (double x, double a0, double a1, double a2, double a3)
{ // 返回  $a_0 + a_1x + a_2x^2 + a_3x^3$ 
  return a0 + (a1 + (a2 + a3 * x) * x) * x;
}

p (x, 7) = 7
p (x, 7, 6) = 19.0018
p (x, 7, 6, 5) = 39.0078
p (x, 7, 6, 5, 4) = 71.0222
```

函数调用 `p(x, a0, a1, a2, a3)` 计算三次多项式  $a_0 + a_1x + a_2x^2 + a_3x^3$  的值。但是，由于 `a1`、`a2`、`a3` 的默认值都是 0，函数也可以通过调用 `p(x, a0)` 来计算常量多项式  $a_0$ 。



或通过调用  $p(x, a_0, a_1)$  来计算一元多项式  $a_0 + a_1x$ , 或通过调用  $p(x, a_0, a_1, a_2)$  来计算二元多项式  $a_0 + a_1x + a_2x^2$ ;

注意, 在函数原型中是如何给定默认值 0 的。例如, 调用  $p(x, 7, 6, 5)$  是与调用  $p(x, 7, 6, 5, 0)$  等价的, 可以计算二元多项式  $7 + 6x + 5x^2$ 。

在上例中, 函数调用时可以使用 2 个、3 个、4 个或 5 个实参, 所以允许默认参数值的结果是允许将不同个数的参数传递给函数。

如果函数具有默认的参数值, 则函数的参数表必须在所有具有默认值的参数右边给出这些参数, 如下所示:

```
void f (int a, int b, int = 4, int = 7, int = 3); // 正确
void g (int a, int = 2, int = 4, int, int = 3); // 错误
```

换句话说, 即必须列出所有“可选的”参数。

## 复 习 题

- 5.1 使用函数实现程序的模块化的好处是什么?
- 5.2 函数声明和函数定义之间的区别是什么?
- 5.3 函数声明可以放在程序中的什么地方?
- 5.4 函数何时需要使用 include 伪指令?
- 5.5 将一个函数定义单独作为一个文件的好处是什么?
- 5.6 单独编译函数的好处是什么?
- 5.7 参数的值传递和引用传递的区别是什么?
- 5.8 参数的引用传递和常量引用传递的区别是什么?
- 5.9 为什么通过值传递的参数是“只读的”, 而通过引用传递的参数是“可读写的”?
- 5.10 下面的函数声明有什么错误:

```
int f (int a, int b=0, int c);
```

## 习 题

- 5.1 在例 5.14 中, 下面的表达式用来测试  $y$  是否是闰年:

```
 $y \% 4 == 0 \&\& y \% 100 != 0 \mid \mid y \% 400 == 0$ 
```

这个表达式不是最有效的形式。如果  $y$  不能被 4 整除, 该表达式仍旧测试一定为假的条件  $y \% 400 == 0$ 。C++ 执行“短路”, 短路的含义是只有需要时才测试组合条件后面的部分。根据短路试写出一个更有效的等价条件。

- 5.2 描述如何将带有一个引用参数的 void 函数转换为带有一个值参数的非 void 函数。

5.3 写出一个与例 5.2 类似的简单程序, 验证三角恒等式  $\cos 2x = 2\cos^2 x - 1$ 。

5.4 写出一个与例 5.2 类似的程序, 验证恒等式:  $\cos^2 x + \sin^2 x = 1$ 。

5.5 写出一个与例 5.2 类似的程序, 验证恒等式:  $b^x = e^{(x \log b)}$ 。

5.6 写出并测试下面的函数, 该函数返回四个给定整数的最小值:

```
int min (int, int, int, int)
```

5.7 写出并测试下面的函数, 该函数使用例 5.5 中的函数 `max (int, int)` 来找到并返回四个给定整数的最大值:

```
int max (int, int, int)
```

5.8 写出并测试下面的函数, 该函数使用函数 `min (int, int)` 来找到并返回四个给定整数的最小值:

```
int min (int, int, int)
```

5.9 写出并测试下面的 `average ()` 函数, 该函数返回四个数的平均值:

```
float average (float x1, float x2, float x3, float x4)
```

5.10 写出并测试下面的 `average ()` 函数, 该函数返回四个正数的平均值:

```
float average (float x1, float x2 = 0, float x3 = 0, float x4 = 0)
```

5.11 使用 `for` 循环实现一个阶乘函数 `fact ()` (参见例 4.9), 确定  $n$  取何值会引起 `fact (n)` 溢出。

5.12 更有效的计算排列函数  $P(n, k)$  的方法是使用公式:

$$P(n, k) = (n)(n-1) \cdots (n-k+2)(n-k+1)$$

即求从  $n$  到  $n-k+1$  的  $k$  个整数的乘积。使用该公式改写并测试例 5.10 中的函数 `perm ()`。

5.13 组合函数  $C(n, k)$  在给定的  $n$  个元素的集合中求不同的 (无序的)  $k$  个元素的子集的个数。该函数可以用以下公式计算:

$$C(n, k) = \frac{n!}{k! (n-k)!}$$

实现该公式。

5.14 组合函数可以用以下公式计算:

$$C(n, k) = \frac{P(n, k)}{k!}$$

使用该公式改写并测试在例 5.13 中的 `comb ()` 函数。

5.15 效率更高的计算  $C(n, k)$  的方法是使用公式

$$C(n, k) = (((((((((n/1)(n-1))/2(n-2))/3) \cdots (n-k+2))/(k-1))(n-k+1))/k$$

公式中交替使用了乘法和除法，每次将从  $n$  开始递减的下一个值相乘，然后除以下一个从 1 开始递增的值。使用该公式改写并测试在例 5.13 中的 `comb()` 函数。

提示：像例 5.12 一样使用 `for` 循环。

5.16 Pascal 三角是开始时像下面一样的三角形数组：

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1

```

在 Pascal 三角中的每个数是一个组合  $C(n, k)$  (参见习题 5.13)。如果对行和列从 0 开始计数，则数字  $C(n, k)$  在  $n$  行  $k$  列。例如，数字  $C(6, 2)$  在第 6 行第 2 列。写出一个程序使用 `comb()` 函数打印 Pascal 三角的前 12 行。

5.17 写出并测试 `digit()` 函数：

```
int digit (int n, int k)
```

该函数返回正整数  $n$  的第  $k$  个数字。例如，如果  $n$  是整数 29 415，则调用 `digit(n, 0)` 会返回数字 5，而调用 `digit(n, 2)` 会返回数字 4。注意数字是从 0 开始数的，并且是从右向左的。

5.18 写出并测试一个实现欧基里德算法的函数，返回两个给定正整数的最大公约数。参见习题 4.14。

5.19 写出并测试一个使用最大公约数函数（习题 5.18）的函数，返回两个给定的正整数的最小公倍数。

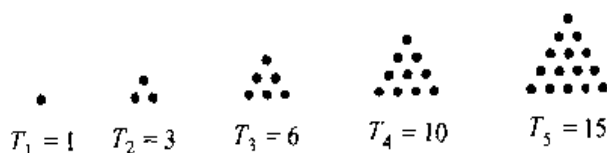
5.20 写出并测试以下的 `power()` 函数，返回  $x$  的幂  $n$ ，这里  $x$  可以为任意整数：

```
double power (double x, int p);
```

使用一个通过将 1 与  $x$  相乘 20 次来计算  $x^{20}$  的算法。

5.21 古希腊的古典几何学。例如，如果卵石博奕的数字可以排列成对称的三角，则该数字称为“三角形的”。前十个三角形的数字是 0, 1, 3, 6, 10, 15, 21, 28, 36 和 45。写出并测试以下的布尔函数：

```
int isTriangular (int n)
```



- 5.22 写出并测试以下的函数 `isSquare()`，判断给定的整数是否是一个平方数：

```
int isSquare (int n)
```

前十个平方数是 0, 1, 4, 9, 16, 25, 36, 49, 64 和 81。

- 5.23 写出并测试以下的函数 `isPentagonal()`，判断给定的整数是否是五边形的：

```
int isPentagonal (int n)
```

前十个五边形的数是 0, 1, 5, 12, 22, 35, 51, 70, 92 和 117。

- 5.24 写出并测试以下的函数 `computeCircle()`，返回一个给定半径  $r$  的圆的面积  $a$  和周长  $c$ ：

```
void computeCircle (float& a, float& c, float r)
```

- 5.25 写出并测试以下的函数 `computeTriangle()`，返回一个给定边长为  $a$ 、 $b$  和  $c$  的三角形的面积  $a$  和周长  $p$ ：

```
void computeTriangle (float& a, float& p, float a, float b, float c)
```

- 5.26 写出并测试以下的函数 `computeSphere()`，返回一个给定半径为  $r$  的球的体积  $v$  和表面积  $s$ ：

```
void computeSphere (float& v, float& s, float r)
```

## 复习题答案

- 5.1 一个单独编译的函数可以看做一个完成特定任务的独立的“黑匣子”。一旦该函数被彻底地测试过，程序员不再需要关心它是如何工作的。这就解放了程序员，使他们可以集中精力进行主程序的开发。另外，如果之后又找到了实现该函数的更好方法，可以在不影响主程序的前提下取代前面的函数。
- 5.2 函数声明（也称为函数原型）只有函数头。函数定义是完整的函数，包括函数头和函数体。函数声明仅提供函数调用所需的信息：函数名，函数的参数类型和返回值；它是函数和调用者之间的接口。函数定义给出有关函数的所有信息，包括函数如何工作的细节；它是函数的实现。
- 5.3 一个函数可以在该函数的所有引用之前的任意位置声明，所以函数声明必须放在调用之前，并且如果函数是单独定义的，则必须将定义放在声明之后。
- 5.4 `include` 伪指令用来包含其他的文件。典型的用法是，函数声明和/或定义放在一个单独的“头”文件中（使用扩展名 `.h`）。如果头文件中仅有函数声明，则函数定义应该在其他文件中单独编译。
- 5.5 将一个函数定义单独作为一个头文件的好处是当修改调用该函数的函数时，不再需要使用编辑器。

- 5.6 单独编译函数的好处是当调用它的函数被再编译时，不需要再次编译该函数。
- 5.7 参数的值传递是复制相应的实参，引用传递仅是重命名相应的实参。
- 5.8 参数常量引用传递不能通过传递它的函数改变。
- 5.9 通过值传递的参数是不能改变的。
- 5.10 该函数中，在参数 (c) 没有默认值之前参数 (b) 有一个默认值。这点违反了以下规定：在函数参数表的其他参数之前的默认参数都必须被列出。

## 习 题 答 案

### 5.1 复合条件

```
y % 4 == 0 && (y % 100 != 0 || y % 400 == 0)
```

是等价且更高效的条件。通过 4 个年份 1995、1996、1900 和 2000 来验证这两个条件的值可以证明它们的等价性。这个复合条件更加高效是因为如果 y 不能被 4 整除（非常可能的情况），则它不再对 y 进行更多的测试。

### 5.2 将引用参数转换为一个返回值。例如，函数：

```
void f (int& n)
{ n * = 2;
}
```

与以下函数等价：

```
int g (int n)
{ return 2 * n;
}
```

两个函数的调用方法是不同的：

```
int x = 22, y = 44;
f (x);    //将 x 的值乘 2
y = g (y); //将 y 的值乘 2
```

但两个函数都是将参数值乘以 2。

### 5.3 下面的程序与例 5.2 类似：

```
int main()
{ for (float x=0; x < 1; x += 0.1)
    cout << cos(2*x) << '\t' << 2*cos(x)*cos(x) - 1 << endl;
}
```

```
1      1
0.980067    0.980067
0.921061    0.921061
0.825336    0.825336
0.696707    0.696707
0.540302    0.540302
```

```
0.362358      0.362358
0.169967      0.169967
-0.0291997    -0.0291997
-0.227202     -0.227202
```

值相等说明对被测试的 10 个  $x$  的值来说, 恒等式是成立的。

#### 5.4 下面的程序与例 5.2 类似:

```
int main()
{ for (double x=0; x < 2; x += 0.2)
  { double s=sin(x);
    double c=cos(x);
    cout << s*s << "\t" << c*c << "\t" << s*s+c*c << endl;
  }
}
```

```
0      1      1
0.0394695    0.96053  1
0.151647     0.848353  1
0.318821     0.681179  1
0.5146  0.4854  1
0.708073     0.291927  1
0.868697     0.131303  1
0.971111     0.028888  1
0.999147     0.000852612  1
0.948379     0.0516208  1
0.826822     0.173178  1
```

#### 5.5 下面的程序与例 5.2 类似:

```
int main()
{ double b=2;
  double lg2=log(2);
  for (double x=0; x < 2; x += 0.2)
    cout << pow(b,x) << "\t" << exp(x*lg2) << endl;
}
```

```
1      1
1.1487  1.1487
1.31951 1.31951
1.51572 1.51572
1.7411  1.7411
2      2
2.2974  2.2974
2.63902 2.63902
3.03143 3.03143
3.4822  3.4822
4      4
```

#### 5.6 下面的函数测试返回四个给定整数的最小值:

```
int min(int,int,int,int);
int main()
{ cout << "Enter four integers: ";
  int w, x, y, z;
```

```

    cin >> w >> x >> y >> z;
    cout << "Their minimum is " << min(w,x,y,z) << endl;
}
int min(int n1, int n2, int n3, int n4)
{ int min=n1;
  if (n2 < min) min = n2;
  if (n3 < min) min = n3;
  if (n4 < min) min = n4;
  return min;
}

```

```

Enter four integers: 44 88 22 66
Their minimum is 22

```

5.7 下面的函数测试返回三个给定整数的最大值:

```

int max(int,int,int);
int main()
{ cout << "Enter three integers: ";
  int x, y, z;
  cin >> x >> y >> z;
  cout << "Their maximum is " << max(x,y,z) << endl;
}
int max(int, int);
int max(int x, int y, int z)
{ int max(int,int);
  return max(max(x,y),z);
}
int max(int x, int y)
{ // 返回两个给定整数的最大值
  if (x < y) return y;
  else return x;
}

```

```

Enter three integers: 44 88 66
Their maximum is 88

```

5.8 下面的函数测试返回四个给定整数的最小值:

```

int min(int,int,int,int);
int main()
{ cout << "Enter four integers: ";
  int w, x, y, z;
  cin >> w >> x >> y >> z;
  cout << "Their minimum is " << min(w,x,y,z) << endl;
}
int min(int,int);
int min(int n1, int n2, int n3, int n4)
{ int m12=min(n1,n2);
  int m34=min(n3,n4);
  return (m12 < m34 ? m12 : m34);
}
int min(int m, int n)
{ return (m < n ? m : n);
}

```

```
Enter four integers: 44 88 22 66
Their minimum is 22
```

5.9 下面的函数测试返回四个数的平均值:

```
double ave(double, double, double, double);
double main()
{ cout << "Enter four numbers: ";
  double w, x, y, z;
  cin >> w >> x >> y >> z;
  cout << "Their average is " << ave(w, x, y, z) << endl;
}
double ave(double x1, double x2, double x3, double x4)
{ return (x1 + x2 + x3 + x4)/4.0;
}
```

```
Enter four numbers: 44 88 22 66
Their average is 55
```

5.10 下面的函数测试返回四个正数的平均值:

```
double ave(double, double=0, double=0, double=0);
double main()
{ cout << "Enter four non-zero numbers: ";
  double w, x, y, z;
  cin >> w >> x >> y >> z;
  cout << "The average of the first one is " << ave(w) << endl;
  cout << "The average of the first two is " << ave(w, x) << endl;
  cout << "The average of the first three is " << ave(w, x, y) << endl;
  cout << "The average of all four is " << ave(w, x, y, z) << endl;
}
double ave(double x1, double x2, double x3, double x4)
{ double sum = x1 + x2 + x3 + x4;
  if (x2 == 0) return sum;
  if (x3 == 0) return sum/2.0;
  if (x4 == 0) return sum/3.0;
  return sum/4.0;
}
```

```
Enter four non-zero numbers: 44 88 22 66
The average of the first one is 44
The average of the first two is 66
The average of the first three is 51.3333
The average of the first four is 55
```

5.11 以下是阶乘函数的测试:

```
long fact(int n);
int main()
{ for (int i=-1; i<20; i++)
  cout << "fact(" << i << ") = " << fact(i) << endl;
}
long fact(int n)
{ if (n < 2) return 1;
```



```

long f=1;
for (int i=2; i <= n; i++)
    f *= i;
return f;
}

```

```

fact (-1) = 1
fact (0) = 1
fact (1) = 1
fact (2) = 2
fact (3) = 6
fact (4) = 24
fact (5) = 120
fact (6) = 720
fact (7) = 5040
fact (8) = 40320
fact (9) = 362880
fact (10) = 3628800
fact (11) = 39916800
fact (12) = 479001600
fact (13) = 1932053504
fact (14) = 1278945280
fact (15) = 2004310016
fact (16) = 2004189184
fact (17) = -288522240
fact (18) = -898433024
fact (19) = 109641728

```

当  $n=13$ ，程序在实现 32 位的 long 类型时溢出。

#### 5.12 以下是排列函数的测试：

```

long perm(int n, int k);
int main()
{ for (int i = -1; i < 6; i++)
  { for (int j = -1; j <= i+1; j++)
    cout << " " << perm(i,j);
    cout << endl;
  }
}
long perm(int n, int k)
{ if (n < 0 || k < 0 || k > n) return 0;
  int p = 1;
  for (int i = 1; i <= k; i++, n--)
    p *= n;
  return p;
}

```

```

0 0
0 1 0
0 1 1 0
0 1 2 2 0
0 1 3 6 6 0
0 1 4 12 24 24 0
0 1 5 20 60 120 120 0

```

## 5.13 以下是组合函数的测试:

```

long comb(int n, int k);
int main()
{ for (int i = -1; i < 6; i++)
  { for (int j = -1; j <= i+1; j++)
    cout << " " << comb(i,j);
    cout << endl;
  }
}
long fact(int n);
long comb(int n, int k)
{ if (n < 0 || k < 0 || k > n) return 0;
  return fact(n)/(fact(k)*fact(n-k));
}
long fact(int n)
{ if (n < 2) return 1;
  long f=1;
  for (int i=2; i <= n; i++)
    f *= i;
  return f;
}

```

```

0 0
0 1 0
0 1 1 0
0 1 2 1 0
0 1 3 3 1 0
0 1 4 6 4 1 0
0 1 5 10 10 5 1 0

```

注意, 函数 fact () 必须在函数 comb () 之前声明, 因为在 comb () 函数中用到了 fact () 函数, 但不需要在 main () 之前声明, 因为在 main () 中没有用到 fact () 函数。

## 5.14 以下是组合函数的测试:

```

long comb(int n, int k);
int main()
{ for (int i = -1; i < 9; i++)
  { for (int j = -1; j <= i+1; j++)
    cout << " " << comb(i,j);
    cout << endl;
  }
}
long perm(int, int);
long fact(int);
long comb(int n, int k)
{ if (n < 0 || k < 0 || k > n) return 0;
  return perm(n,k)/fact(k);
}
long perm(int n, int k)
{ if (n < 0 || k < 0 || k > n) return 0;

```

```

    int p = 1;
    for (int i = 1; i <= k; i++, n--)
        p *= n;
    return p;
}
long fact(int n)
{ if (n < 2) return 1;
  long f=1;
  for (int i=2; i <= n; i++)
      f *= i;
  return f;
}

```

输出与习题 5.13 相同。

#### 5.15 以下是组合函数的测试:

```

long comb(int n, int k);
int main()
{ for (int i = -1; i < 9; i++)
  { for (int j = -1; j <= i+1; j++)
    cout << " " << comb(i,j);
    cout << endl;
  }
}
long comb(int n, int k)
{ if (n < 0 || k < 0 || k > n) return 0;
  long c = 1;
  for (int i=1; i<=k; i++, n--)
      c = c*n/i;
  return c;
}

```

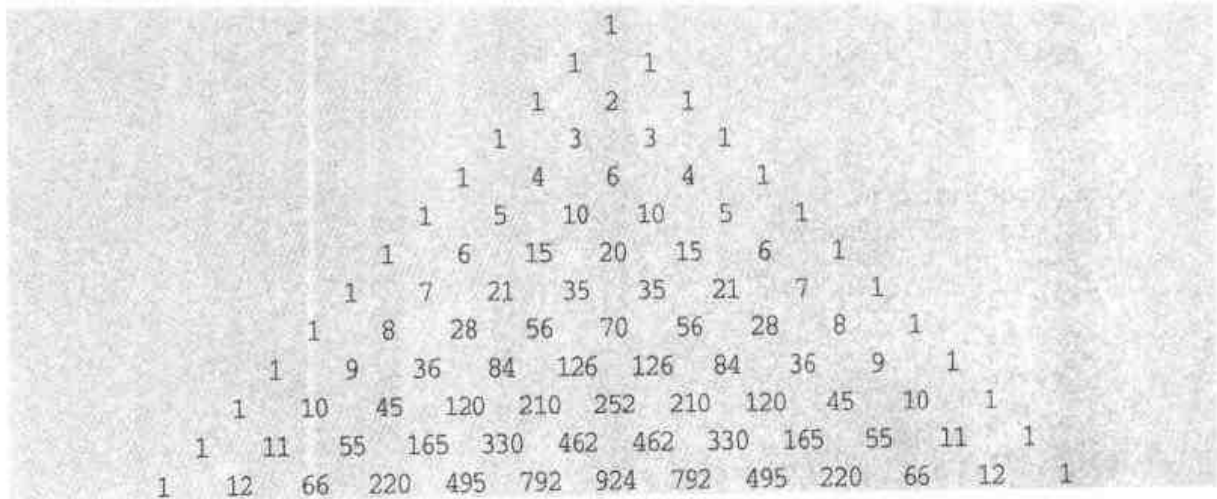
输出与习题 5.13 相同。

#### 5.16 以下程序打印 Pascal 三角:

```

long comb(int n, int k);
int main()
{ const m = 13;
  for (int i = 0; i < m; i++)
  { for (int j = 1; j < m-i; j++)
    cout << setw(2) << " "; // print whitespace
    for (int j = 0; j <= i; j++)
        cout << setw(4) << comb(i,j);
    cout << endl;
  }
}
long comb(int n, int k)
{ if (n < 0 || k < 0 || k > n) return 0;
  long c = 1;
  for (int i=1; i<=k; i++, n--)
      c = c*n/i;
  return c;
}

```



5.17 以下是从一个整数中提取数字的函数的测试:

```
int digit(long, int);
int main()
{ int n, k;
  cout << "Integer: ";
  cin >> n;
  do
  { cout << "Digit: ";
    cin >> k;
    cout << "Digit number " << k << " of " << n
      << " is " << digit(n, k) << endl;
  } while (k > 0);
};

int digit(long n, int k)
{ for (int i = 0; i < k; i++)

    n /= 10; // 去掉最右边的数字
  return n % 10;
}
```

```
Integer: 876543210
Digit: 4
Digit number 4 of 876543210 is 4
Digit: 7
Digit number 7 of 876543210 is 7
Digit: 0
Digit number 0 of 876543210 is 0
```

5.18 以下是求最大公约数的函数的测试:

```
long gcd(long, long);
int main()
{ int m, n;
  cout << "Enter two positive integers: ";
  cin >> m >> n;
  cout << "gcd(" << m << ", " << n << ") = " << gcd(m, n) << endl;
}

long gcd(long m, long n)
```

```

{ // 返回 m 和 n 的最大公约数
  if (m<n) swap(m,n);
  assert(n >= 0);
  while (n>0)
  { long r=m%n;
    m = n;
    n = r;
  }
  return m;
}

```

Enter two positive integers: **144 192**  
gcd (144, 192) = 48

### 5.19 以下是求最小公倍数的函数的测试:

```

long lcm(long, long);
int main()
{ int m, n;
  cout << "Enter two positive integers: ";
  cin >> m >> n;
  cout << "lcm(" << m << ", " << n << ") = " << lcm(m,n) << endl;
}
long gcd(long, long);
long lcm(long m, long n)
{ return m*n/gcd(m,n);
}
long gcd(long m, long n)
{ if (m < n) swap(m,n);
  while (n>0)
  { int r = m%n;
    m = n;
    n = r;
  }
  return m;
}

```

Enter two positive integers: **144 192**  
lcm (144, 192) = 576

### 5.20 以下是幂函数的测试:

```

double pow(double, int);
int main()
{ cout << "Enter a positive float x and an integer n: ";
  double x;
  int n;
  cin >> x >> n;
  cout << "pow(" << x << ", " << n << ") = " << pow(x,n) << endl;
}
double pow(double x, int n)
{ if (x == 0) return 0;
  if (n == 0) return 1;
  double y=1;

```

```

    for (int i=0; i < n; i++)
        y *= x;
    for (int i=0; i > n; i--)
        y /= x;
    return y;
}

```

Enter a positiv float x and an integer n: 2.0 -3  
 pow(2, -3) = 0.125

5.21 以下是测试整数的三角形性的布尔函数的测试:

```

int isTriangular(int);
int main()
{ const int MAX=12;
  for (int i=0; i<MAX; i++)
    if (isTriangular(i)) cout << i << " is triangular.\n";
    else cout << i << " is not triangular.\n";
}
int isTriangular(int n)
{ int x=0, y=0, dy=1;
  while (y < n)
    y += dy++;
  if (y == n) return true;
  else return false;
}

```

```

0 is triangular.
1 is triangular.
2 is not triangular.
3 is triangular.
4 is not triangular.
5 is not triangular.
6 is triangular.
7 is not triangular.
8 is not triangular.
9 is not triangular.
10 is triangular.
11 is not triangular.

```

5.22 以下是测试整数是否为平方数的布尔函数的测试:

```

int isSquare(int);
int main()
{ const int MAX=20;
  for (int i=0; i<MAX; i++)
    if (isSquare(i)) cout << i << " is square.\n";
    else cout << i << " is not square.\n";
}
int isSquare(int n)
{ int i=0;
  while (i*i<n)
    ++i;
  if (i*i == n) return true;
  else return false;
}

```

```

0 is square.
1 is square.
2 is not square.
3 is not square.
4 is square.
5 is not square.
6 is not square.
7 is not square.
8 is not square.
9 is square.
10 is not square.
11 is not square.

```

5.23 以下是测试整数是否为五边形的布尔函数的测试:

```

int isPentagonal(int);
int main()
{ const int MAX=40;
  for (int i=0; i<MAX; i++)
    if (isPentagonal(i)) cout << i << " is pentagonal.\n";
    else cout << i << " is not pentagonal.\n";
}
int isPentagonal(int n)
{ int x=0, y=0, dy=1;
  while (y < n)
  { y += dy;
    dy += 3;
  }
  if (y == n) return true;
  else return false;
}

```

```

0 is pentagonal.
1 is pentagonal.
2 is not pentagonal.
3 is not pentagonal.
4 is not pentagonal.
5 is pentagonal.
6 is not pentagonal.
7 is not pentagonal.
8 is not pentagonal.
9 is not pentagonal.
10 is not pentagonal.
11 is not pentagonal.
12 is pentagonal.
13 is not pentagonal.

```

5.24 以下测试一个带有引用参数的函数:

```

void computeCircle(double& area, double& circ, double r);
int main()
{ double a, c, r;
  cout << "Enter the radius: ";
  cin >> r;

```

```

    computeCircle(a,c,r);
    cout << "The area of a circle of radius " << r << " is " << a
        << "\nand its circumference is " << c << endl;
}
void computeCircle(double& area, double& circ, double r)
{ const double PI=3.141592653589793;
  area = PI*r*r;
  circ = 2*PI*r;
}

```

Enter the radius: 10  
 The area of a circle of radius 10 is 314.159  
 and its circumference is 62.8319

### 5.25 以下测试一个带有引用参数的函数:

```

void computeTriangle(float& a, float& p, float x, float y, float z);
int main()
{ float a, p, x, y, z;
  cout << "Enter the sides: ";
  cin >> x >> y >> z;
  computeTriangle(a,p,x,y,z);
  cout << "The area of the triangle is " << a
      << "\nand its perimeter is " << p << endl;
}
void computeTriangle(float& a, float& p, float x, float y, float
z)
{ p = x + y + z;
  float s = p/2.0; // 三角形的半周长
  a = sqrt(s*(s-x)*(s-y)*(s-z)); // 海伦公式
}

```

Enter the sides : 30 50 40  
 The area of the triangle is 600  
 and its perimeter is 120

### 5.26 以下测试一个带有引用参数的函数:

```

void computeSphere(double& a, double& v, double r);
int main()
{ double a, v, r;
  cout << "Enter the radius: ";
  cin >> r;
  computeSphere(a,v,r);
  cout << "The area of a sphere of radius: " << r << " is " << a
      << "\nand its volume is " << v << endl;
}
void computeSphere(double& a, double& v, double r)
{ const double PI=3.141592653589793;
  a = 4.0*PI*r*r;
  v = a*r/3.0;
}

```

Enter the radius: 10  
 The area of a sphere of radius 10 is 1256.64  
 and its volume is 4188.79



# 第 6 章 数 组

## 6.1 介绍

数组是类型相同的对象的序列。这里的对象称为数组元素，并被按顺序编号为 0, 1, 2, 3, …, 这些编号称为数组的下标。这里使用“下标”是因为作为一个数字的序列，数组可以使用下标写成下面的形式： $a_0, a_1, a_2, \dots$ 。数组下标指定数组元素在数组中的位置，由此可以直接实现数组的随机存取。

如果数组名为  $a$ ，则在位置 0 的数组元素名为  $a[0]$ ，在位置 1 的数组元素名为  $a[1]$ ，依此类推。一般而言，第  $i$  个元素的位置是  $i - 1$ 。所以，如果数组有  $n$  个元素，它们的名字为  $a[0], a[1], a[2], \dots, a[n - 1]$ 。

可以将数组想像成一连串的用下标值编号的相邻存储区。例如，图 6.1 用来表示一个名字为  $a$  的有 5 个元素的数组， $a[0]$  中存放 11.11， $a[1]$  中存放 33.33， $a[2]$  中存放 55.55， $a[3]$  中存放 77.77， $a[4]$  中存放 99.99，这张表事实上代表了计算机内存中的一段区域，因为在内存中总是按照元素的相邻关系存放数组的。

$a$	
0	11.11
1	33.33
2	55.55
3	77.77
4	99.99

将第  $i$  个元素编号为下标  $i - 1$  的方法称为“从 0 开始的下标”，这样可以保证每个数组元素的下标与该元素到第一个元素  $a[0]$  的“步数”相等。例如，元素  $a[3]$  到  $a[0]$  的步数为 3 步。

图 6.1 数组  $a$  示例

实际上所有有用的程序都用到数组，如果要用同样的方法使用一些具有相同类型的对象时，通常比较简单的方法是将这些对象组成一个数组。

## 6.2 数组的处理

数组是一个复合对象：它由具有独立值的若干元素组成。相对地，一个具有基本类型的普通变量被称为矢量对象。

本章的第一个例子说明数组元素的赋值和访问与一个普通变量一样。

### 例 6.1 数组的随机存取

```
int main ()
{ double a [3];
  a [2] = 55.55;
```

```

a[0] = 11.11;
a[1] = 33.33;
cout << "a[0] = " << a[0] << endl;
cout << "a[1] = " << a[1] << endl;
cout << "a[2] = " << a[2] << endl;
|
a[0] = 11.11
a[1] = 33.33
a[2] = 55.55

```

其中，第一个语句声明了一个类型为 `double` 的有 3 个元素的数组，下面的 3 个语句给数组赋值。

数组通常使用 `for` 循环来处理。

### 例 6.2 按照顺序打印一个数的序列

本例读入 5 个数字，然后按照相反的顺序打印它们：

```

int main ()
{
    const int SIZE=5;    // 定义 n 的大小为 5 个元素
    double a [SIZE];     // 把数组的元素声明为双精度类型
    cout << "Enter " << SIZE << " numbers: \t";
    for (int i=0; i<SIZE; i++)
        cin >> a[i];
    cout << "In reverse order: ";
    for (int i=SIZE-1; i>=0; i--)
        cout << "\t" << a[i];
}

```

Enter 5 numbers:    11.11   33.33   55.55   77.77   99.99  
 In reverse order:   99.99   77.77   55.55   33.33   11.11

其中，第一条语句定义了一个值为 5 的符号常量 `SIZE`，第二个语句声明了一个类型为 `double` 的有 5 个元素的数组 `a`，然后第一个 `for` 循环将 5 个值读入数组，而第二个 `for` 循环以相反的顺序打印它们。

数组声明的语法为：

类型    数组名 [数组大小];

其中，类型是数组元素的类型，数组大小是数组元素的个数。例 6.1 的数组声明为：

```
double a [SIZE];
```

声明了有 5 个元素的数组 `a`，每个元素的类型都是 `double`。标准的 C++ 要求数组大小是一个正整数常量，所以数组大小必须是一个像例 6.1 中一样的符号常量，或是一个整数，如下所示：

```
double a [5];
```

通常，由于在处理数组的 `for` 循环中使用的是同一个数组大小的值，所以最好使用符号

常量。

## 6.3 数组的初始化

在 C++ 中, 可以使用一个可选的初始化列表来初始化一个数组, 如下所示 (示例见图 6.2):

```
float a [] = {22.2, 44.4, 66.6};
```

a	
0	22.2
1	44.4
2	66.6

图 6.2 数组 a 示例

列表中的值按顺序赋值给数组元素, 数组的大小要与初始化表中的值的个数相等。所以上面的代码声明了一个有 3 个元素的 float 数组 a, 然后, 使用在初始化表中给定的 3 个值初始化这 3 个元素如图 6.2 所示。

### 例 6.3 数组的初始化

本例初始化数组, 然后打印它的值:

```
int main ()
{ float a [] = { 22.2, 44.4, 66.6 };
  int size = sizeof (a) /sizeof (float);
  for (int i=0; i<size; i++)
    cout << "\ta[" << i << "] = " << a[i] << endl;
```

```

a[0] = 22.2
a[1] = 44.4
a[2] = 66.6
```

第一个语句声明了一个如上所述的有 3 个元素的数组, 第二个语句使用 sizeof () 函数计算在数组中的元素的实际个数。sizeof (float) 的值为 4, 因为在这个机器上一个 float 值占用内存的 4 个字节。sizeof (a) 的值为 12, 因为整个数组在内存中占用 12 个字节。因此, size 的值是  $12/4=3$ 。

通过使用确定的数组大小值, 同时带有初始化列表来声明一个数组:

```
float a [7] = {55.5, 66.6, 77.7};
```

a	
0	55.5
1	66.6
2	77.7
3	0.0
4	0.0
6	0.0
7	0.0

图 6.3 数组 a 示例

该数组声明有 7 个 float 类型的元素, 然后, 数组的初始化列表使用给定值对前 3 个元素进行初始化, 而将剩余的 4 个元素初始化为 0 (如图 6.3 所示)。

### 例 6.4 数组初始化

本例初始化数组, 然后打印它的值:

```
int main ()
{ float a [7] = { 22.2, 44.4, 66.6 };
```

```
int size = sizeof(a) / sizeof(float);
for (int i=0; i<size; i++)
    cout << "\ta[" << i << "] = " << a[i] << endl;
}
```

```
a[0] = 22.2
a[1] = 44.4
a[2] = 66.6
a[3] = 0
a[4] = 0
a[5] = 0
a[6] = 0
```

注意，在数组的初始化列表中的值的个数不能超过数组的大小：

```
float a[3] = {22.2, 44.4, 66.6, 88.8}; // 错误：太多的值！
```

可以使用一个空的初始化列表将数组初始化为全 0。因此，下面的三个声明是等价的：

```
float a[] = {0, 0, 0, 0, 0, 0, 0, 0, 0};
float a[9] = {0, 0};
float a[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0};
```

但是，注意它们与没有初始化列表的数组声明不同。正如一个基本类型的变量，如果一个数组没有初始化，它将包含“垃圾”值。

### 例 6.5 一个未初始化的数组

本例初始化一个数组然后打印它的值：

```
int main()
{ const int SIZE=4; // 定义 N 的大小为 4 个元素
  float a[SIZE]; // 把数组的元素声明为浮点类型
  for (int i=0; i<SIZE; i++)
      cout << "\ta[" << i << "] = " << a[i] << endl;
}
```

```
a[0] = 6.01838e-39
a[1] = 9.36651e-39
a[2] = 6.00363e-39
a[3] = 0
```

注意一个未被初始化的数组的值可能是 0，也可能不是 0；这依赖于在程序执行前存放数组的那部分内存是如何使用的。

注意初始化与赋值不同。数组可以初始化，但是不能被赋值：

```
float a[7] = {22.2, 44.4, 66.6};
float b[7] = {33.3, 55.5, 77.7};
b=a; // 错误：数组不能被赋值！
```

一个数组也不能使用另一个数组进行初始化：

```
float a[7] = {22.2, 44.4, 66.6};
float b[7] = a; // 错误：数组不能用做初始化器！
```

## 6.4 数组元素下标越界

在某些程序设计语言中，一个下标变量不允许超出数组定义中所设的界限。例如，在 Pascal 中，如果一个数组 a 的下标被定义为从 0 到 3，则引用 a[6] 将使数组崩溃。在 C++（或 C）的数组中是没有这种安全措施的。正如下一个例子所示，下标变量可以远远超出它的定义范围，而计算机却不能测试出任何错误。

### 例 6.6 允许数组的下标越界

本例中有一个运行时错误：它访问了没有分配给它的一部分内存。

```
int main ()
{ const int SIZE = 4;
  float a[SIZE] = { 33.3, 44.4, 55.5, 66.6 };
  for (int i = 0; i < 7; i++) // 错误：下标超出范围
    cout << "\ta[" << i << "] = " << a[i] << endl;
```

```

a[0] = 33.3
a[1] = 44.4
a[2] = 55.5
a[3] = 66.6
a[4] = 5.60519e-45
a[5] = 6.01888e-39
a[6] = 6.01889e-39
```

最后被打印的 3 个值是无用的值，是以前使用这些内存单元时遗留下的值。

允许一个数组的下标越界会产生很严重的副作用，如下例所示

### 例 6.7 引起副作用

本例在访问数组的一个不存在的元素时无意中改变了一个变量的值：

```
int main ()
{ float a[] = { 22.2, 44.4, 66.6 };
  float x = 11.1;
  cout << "x = " << x << endl;
  a[3] = 88.8; // 错误：下标超出范围
  cout << "x = " << x << endl;
}

x = 11.1
x = 88.8
```

变量 x 是在数组 a 之前声明的，所以系统将 4 字节的内存区域分配给 x，然后将紧跟在该区域后面的 12 个字节分配给 a 的 3 个元素。因此，a 和 x 占用了内存中连续的 16 字节，就好像 x 是 a[3]。所以当程序将 88.8 赋值给 a[3]（是不存在的）时，实际上将 x 的

值修改为 88.8。这种情况如图 6.4 所示，图中表示了内存中 20 个连续的字节；存放 88.8 的 4 个字节紧跟在存放 66.6 的 4 个字节之后。

这是一种最糟糕的运行时错误，它修改了一个完全独立的变量的值，而在代码中甚至没有提及这个错误。这种错误称为副作用，由于它可能不被察觉，所以会造成惨重的损失。

C++ 程序员有责任保证数组的下标值保持在范围之内。如例 6.7 所示，如果所引起的副作用不可预测，则逃避这种责任的后果是非常严重的。

### 例 6.8 引起不能处理的异常

由于数组下标太大而引起本程序崩溃：

```
int main ()
{ const int SIZE= 4;
  float a [] = { 22.2, 44.4, 66.6 };
  float x= 11.1;
  cout << "x = " << x << endl;
  a [3333] = 88.8;    // 错误：下标超出范围
  cout << "x = " << x << endl;
  cout << a << endl;
}
```

当在一个 Windows 工作站中运行时，本程序会产生一个如图 6.5 所示的警告框。该框提示程序企图访问地址为 0040108e 的内存单元，这个单元在分配给运行该程序的进程的内存区域之外，所以 Windows 操作系统终止该程序。

在例 6.8 中的运行时错误称为不能处理的异常，因为在程序中没有处理这种错误的代码。可以在 C++ 程序中包含保证程序不会崩溃的代码，这种代码称为异常处理代码。

与一些其他的编程语言不同（如 Pascal 和 Java），标准的 C++ 编译器将不允许数组赋值，并且不限制数组下标越界，防止编译错误和运行时错误是程序员的责任。这样做的好处是代码运行速度更快，效率更高。如果这些好处对于你的程序来讲并不重要，则可以使用标准 C++ 的 vector 对象来代替数组（参见第 10 章）。

## 6.5 将数组传递给函数

在前面的例子中声明数组 a 的代码 float a [] 告诉编译器两件事：数组名为 a，并且数组元素的类型为 float。符号 a 存放了数组的内存地址，所以代码提供给编译器声明数组所需的所有信息，数组的大小（即数组中元素的个数）不需要告诉编译器。当将一个数组作为参

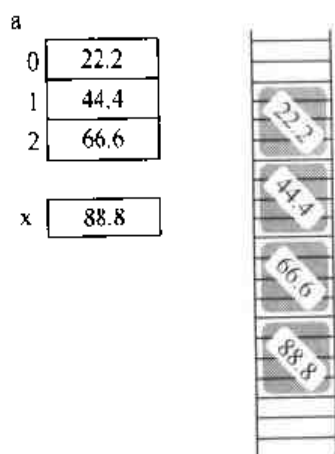


图 6.4 数组 a 与变量 x 在内存中的位置



图 6.5 错误画面

数传递给函数时，C++ 编译器需要相同的信息。

### 例 6.9 将一个数组传递给一个返回其和的函数

```
int sum (int [], int);
int main ()
{ // 测试使用数组参数
  int a [] = { 11, 33, 55, 77 };
  int size = sizeof (a) / sizeof (int);
  cout << "sum (a, size) = " << sum (a, size) << endl;
}
int sum (int a [], int n)
{ int sum=0;
  for (int i=0; i<n; i++)
    sum += a [i];
  return sum;
}
```

~~sum (a, size) = 176~~

函数的参数表是 (int a [], int n)，在 main () 上面用来声明函数的函数原型的形式为 (int [], int); 除了表示函数名的参数被删去，它与参数表是相同的。在 main () 中的函数调用的代码是 sum (a, size); 其中列出不带类型的参数。注意对象 a 的类型为 int []。

当一个数组被传递给一个函数时，如例 6.9 中的函数调用 sum (a, size)，数组名 a 的值实质上是数组的第一个元素 (a [0]) 的内存地址。函数通过这个地址访问和修改数组的内容。所以将一个数组传递给函数与通过引用传递一个变量相同；函数可以修改数组元素的值，如下例所示。

### 例 6.10 数组的输入和输出函数

本例使用一个 read () 函数采用交互方式将值输入数组，然后使用 print () 函数打印该数组：

```
void read (int [], int&);
void print (int [], int);
int main ()
{ const int MAXSIZE=100;
  int a [MAXSIZE] = {0}, size;
  read (a, size);
  cout << "The array has " << size << " elements: ";
  print (a, size);
}
void read (int a [], int&n)
{ cout << "Enter integers. Terminate with 0: \n";
  n=0;
  do
  { cout << "a [" << n << "]: ";
    cin >> a [n];
  } while (a [n++] != 0 && n< MAXSIZE);
}
```

```

--n;    //不计这个0
|
void print (int a [], int n)
| for (int i=0; i<n; i++)
    cout << a[i] << " ";
|
Enter integers. Terminate with 0:
a [0]: 11
a [1]: 22
a [2]: 33
a [3]: 44
a [4]: 0
The array has 4 elements: 11 22 33 44

```

read () 函数修改数组 a 和数组大小 n 的值。由于 n 是一个矢量变量，必须通过引用传递以允许函数修改它的值。由于 a 是一个数组变量，必须通过值传递以使函数可以修改数组元素的值。

注意数组的大小必须显式地传递给处理它的函数，在 C++ 中一个函数不能计算传递给它的数组的大小。

例 6.10 说明，即使数组变量是通过值传递的，函数也可以修改数组元素的值。这是可能的，因为数组变量本身是数组第一个元素的内存地址，传递该地址给函数，函数就得知了它访问和修改存放数组的内存区域所需的全部信息，这是通过给定的内存地址和数组下标计算数组元素的位置来实现的。例如，在例 6.10 中的输入语句

```
cin >> a[n];
```

当  $n=3$  时，系统计算出  $a[3]$  的内存地址在  $a[0]$  的内存地址后面的 12 字节 ( $3 \times 4 = 12$ )，这个地址在变量 a 中通过值传递给函数，所以函数获得  $a[3]$  的确切地址。例如，假设  $a[0]$  存放在从地址 0x0064fdbc (“0x0064fdbc” 是一个十六进制数，值为 6 618 556) 开始的 4 个连续的字节中，则经过计算  $a[3]$  的地址为 0x0064fdc8 (6 618 568 = 6 618 556 + 12)。计算中，数值 12 称为对元素  $a[3]$  的偏移量 (参见问题 6.5 及附录 G 中有关十六进制的信息)，如图 6.6 所示。

注意数组名 (其值是一个内存地址) 本身是一个常量，所以不能在任何地方修改，这表示数组不能移动到内存中的另一个地方。

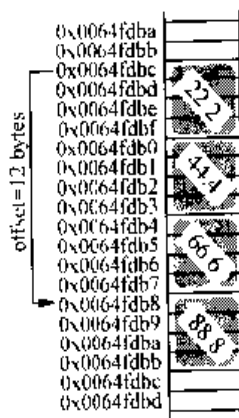


图 6.6 偏移量示例

### 例 6.11 打印一个数组的内存位置

本例打印存储在数组名中的地址的值：

```

int main ()
| int a [] = { 22, 44, 66, 88 };
    cout << "a = " << a;    // a [0] 的地址
|
a = 0x0064fdb8

```



数组名 `a` 有两个解释。当它与下标一起使用来命名一个数组元素时, 它确定一个完整的复合对象。但是, 作为一个变量, 它的值是数组的第一个元素 `a[0]` 的第一个字节的内存地址(如图 6.7 所示)。

a	
0	22.2
1	44.4
2	66.6
3	88.8

a 0x0064fdb0

图 6.7 数组的地址

## 6.6 线性查找算法

相对于其他用途, 大概计算机更多的用途是存取信息。数据经常使用类似于数组的有序结构存储。查找一个对象的最简单的方法是从数组的第一个元素开始, 一个挨一个地检查每一个元素, 直到找到所需的对象。这种方法称为线性查找算法。

### 例 6.12 线性查找

本例测试一个实现线性查找算法的函数:

```
int index (int, int [], int);
int main ()
{ int a [] = { 22, 44, 66, 88, 44, 66, 55 };
  cout << "index (44, a, 7) = " << index (44, a, 7) << endl;
  cout << "index (50, a, 7) = " << index (50, a, 7) << endl;
}

int index (int x, int a [], int n)
{ for (int i = 0; i < n; i++)
  if (a [i] == x) return i;
  return n; //x 未找到
}

index (44, a, 7) = 1
index (50, a, 7) = 7
```

## 6.7 冒泡排序算法

线性查找算法的效率不高, 在电话号码簿中查找一个名字时这显然不是一个好的方法。可以以更高的效率完成这个任务, 因为电话号码簿中的名字是以字母顺序存放的。为了在类似数组的有序的数据结构上使用一个更高效的算法, 首先必须将该结构排序, 使其中的元素按顺序排列。

有许多算法可以对数组进行排序。虽然并不像其他的算法那样效率高, 冒泡排序是一种最简单的排序算法。它通过一系列的循环操作, 每次将下一个最大值移动到正确的位置。在每次循环中, 它比较每一对相邻的元素对, 将其中较大的元素移到上面。

### 例 6.13 冒泡排序

本例测试一个实现冒泡排序算法的函数。

```

void print (float [], int);
void sort (float [], int);
int main ()
{ float a [] = {55.5, 22.5, 99.9, 66.6, 44.4, 88.8, 33.3, 77.7};
  print (a, 8);
  sort (a, 8);
  print (a, 8);
}
void sort (float a [], int n)
{ // 冒泡排序
  for (int i=1; i<n; i++)
    // 排出最大值 max {a [0..n-i]}:
    for (int j=0; j<n-i; j++)
      if (a [j] > a [j+1]) swap (a [j], a [j+1]);
    // 不变量 a [n-1-i..n-1] 是排序的
}
55.5, 22.5, 99.9, 66.6, 44.4, 88.8, 33.3, 77.7
22.5, 33.3, 44.4, 55.5, 66.6, 77.7, 88.8, 99.9

```

sort () 函数使用了两个嵌套的循环。内层的 for 循环比较每一对相邻的元素，并且当它们的顺序不满足要求时进行交换。通过这种方法，使每个元素经过所有小于它的元素“起泡浮起”。

## 6.8 二分查找算法

二分查找使用了“分开和战胜”策略。它反复的将数组分为两部分，然后在可能包含目标值的那部分中进行查找。

### 例 6.14 二分查找算法

本例测试实现二分查找算法的函数。它使用了在例 6.12 中测试线性查找算法相同的测试程序：

```

int index (int, int [], int);
int main ()
{ int a [] = { 22, 33, 44, 55, 66, 77, 88 };
  cout << "index (44, a, 7) = " << index (44, a, 7) << endl;
  cout << "index (60, a, 7) = " << index (60, a, 7) << endl;
}
int index (int x, int a [], int n)
{ // 先决条件 a [0] <= a [1] <= ... <= a [n-1];
  // 二分查找
  int lo=0, hi=n-1, i;
  while (lo <= hi)
  { i = (lo + hi) / 2;
    if (a [i] == x) return i;
    if (a [i] < x) lo = i+1;    // 在 a [i+1..hi] 中连续查找
    else hi = i-1;            // 在 a [lo..i-1] 中连续查找
  }
}

```

```

    }
    return n;    // 没有在 a [0..n-1] 中发现 x
}
index (44, a, 7) = 2
index (60, a, 7) = 7

```

注意在使用二分算法之前数组已经排好序，这个要求是在函数代码中的注释 PRECONDITION 中表达的。

在 while 循环的每次循环中，检查子数组  $a[lo..hi]$ （即从  $a[lo]$  到  $a[hi]$  之间的所有元素）的中间元素，如果它不是目标元素，则继续检查较大的一半数组  $a[i+1..hi]$  或较小的一半数组  $a[lo..i-1]$ 。如果  $(a[i] < x)$ ，则  $x$  不可能在较小的一半中（因为数组是按照递增的顺序排列的），所以较小的一半可以被忽略，只要继续检查较大的一半即可。同理，如果条件  $(a[i] < x)$  为假，则只要继续检查较小的一半即可。因此在每次循环中，查找的范围减小 50%。当在  $a[i]$  找到  $x$  时函数返回，或当  $lo > hi$  时循环终止。在后一种情况下，子数组  $a[lo..hi]$  为空，表示没找到  $x$ ，所以函数返回  $n$ 。

图 6.8 表示了调用  $index(44, a, 7)$  的执行过程。当循环开始时， $x = 44$ ， $n = 7$ ， $lo = 0$ ， $hi = 6$ ；数组  $a[0..6]$  中间的元素是  $a[3] = 55$ ，它大于  $x$ ，所以将  $hi$  重新设置为  $i - 1 = 2$ 。在第二次循环中， $lo = 0$  且  $hi = 2$ ；子数组  $a[0..2]$  中间的元素是  $a[1] = 33$ ，它小于  $x$ ，所以将  $lo$  重新设置为  $i + 1 = 2$ 。在第三次循环中， $lo = 2$  且  $hi = 2$ ；子数组  $a[2..2]$  的中间元素是  $a[2] = 44$ ，它与  $x$  相等，所以函数返回 2，表示目标  $x$  是在  $a[2]$  中找到的。

lo	hi	i	a[i]	??	x
0	6	3	55	>	44
	2	1	33	<	44
2		2	44	=	44

图 6.8 调用  $index(44, a, 7)$  时的变化

图 6.9 表示了调用  $index(60, a, 7)$  的执行过程。当循环开始时， $x = 44$ ， $n = 7$ ， $lo = 0$ ， $hi = 6$ ；数组  $a[0..6]$  中间的元素是  $a[3] = 55$ ，它小于  $x$ ，所以将  $lo$  重新设置为  $i + 1 = 4$ 。在第二次循环中， $lo = 4$  且  $hi = 6$ ；子数组  $a[4..6]$  中间的元素是  $a[5] = 77$ ，它大于  $x$ ，所以将  $hi$  重新设置为  $i - 1 = 4$ 。在第三次循环中， $lo = 4$  且  $hi = 4$ ；子数组  $a[4..4]$  的中间元素是  $a[4] = 66$ ，它大于  $x$ ，所以将  $hi$  重新设置为  $i - 1 = 3$ ，这将终止循环，所以函数返回 7，表示目标  $x$  没有找到。

lo	hi	i	a[i]	??	x
0	6	3	55	>	60
4		5	77	<	60
	4	4	66	>	60

图 6.9 调用  $index(60, a, 7)$  时的变化

二分查找算法比线性查找算法要好的多。两者最重要的区别是二分查找仅对排好序的数组进行操作，这样做的好处是二分算法比线性查找算法的速度要快得多。例如，对一个有 100 个元素的数组，线性查找算法需要做 100 次循环，但二分查找算法不论目标是什么，只需要不超过 8 次的循环，这是因为二分查找算法以对数时间运行，即循环的次数不会超过  $\lg n + 1$ ，其中  $n$  为数组的大小，而  $\lg n$  是  $n$  的以 2 为底的对数。当  $n = 100$ ， $\lg n + 1 = 7.64$ 。

注意在例 6.14 中,  $n=7$  个元素, 所以  $\lg n + 1 = 3.81$ ; 这表示所需的循环最多不超过 3 次。

两个算法的第二个区别是线性查找算法返回满足条件  $a[i] = x$  的最小的下标  $i$ , 但是二分查找算法不是这样: 如果在数组中有多个  $x$ , 不能确定返回的下标是哪个  $x$  的位置。

因为二分查找算法要求数组必须是有序的, 所以有一个单独函数来测试这个条件就非常有用。

### 例 6.15 判断一个数组是否有序

下例测试一个判断给定数组是否有序的布尔函数。

```
bool isNondecreasing (int a [], int n);
int main ()
{ int a [] = { 22, 44, 66, 88, 44, 66, 55 };
  cout << "isNondecreasing (a, 4) = " << isNondecreasing (a, 4) << endl;
  cout << "isNondecreasing (a, 7) = " << isNondecreasing (a, 7) << endl;
}
bool isNondecreasing (int a [], int n)
{ // 当且仅当  $a[0] \leq a[1] \leq \dots \leq a[n-1]$ , 返回 true
  for (int i=1; i<n; i++)
    if (a[i] < a[i-1]) return false;
  return true;
}
isNondecreasing (a, 4) = 1
isNondecreasing (a, 7) = 0
```

如果函数找到任意一个相邻的元素对  $(a[i-1], a[i])$  是递减排列的 (即  $a[i] < a[i-1]$ ), 则返回 false; 否则, 返回 true, 表示数组不是递减的。

注意布尔值是被打印成整数值 1 和 0 的, 也就是它们在内存中所存的值。

如果例 6.14 中数组有序的前提条件为假, 二分查找算法 `search()` 将不能正确执行。使用在头文件 `<cassert>` 中定义的 `assert()` 函数可以自动检查这个条件。该函数使用一个布尔型的实参, 如果该实参为 false, 则函数终止程序并将这件事报告给操作系统。如果该实参为 true, 则程序执行不受影响。

### 例 6.16 使用 `assert()` 函数加强前提条件。

下例测试改进的 `search()` 函数。该函数使用例 6.15 中的 `isNondecreasing()` 函数判断数组是否是有序的, 它将布尔型的结果返回给 `assert()` 函数, 所以如果数组不是有序的, 则查找将不会执行。

```
#include <cassert> // 定义 assert() 函数
#include <iostream> // 定义 cout 对象
using namespace std;
int index (int x, int a [], int n);
int main ()
{ int a [] = { 22, 33, 44, 55, 66, 77, 88, 60 };
  cout << "index (44, a, 7) = " << index (44, a, 7) << endl;
  assert (isNondecreasing (a, 8));
}
```

```

    cout << "index (44, a, 7) = " << index (44, a, 8) << endl;
    cout << "index (60, a, 7) = " << index (60, a, 8) << endl;
}
bool isNondecreasing (int a [], int n);
int index (int x, int a [], int n)
// 先决条件 a [0] <= a [1] <= ... <= a [n-1];
// 二进制查找
assert (isNondecreasing (a, n));
int lo = 0, hi = n-1, i;
while (lo <= hi)
{ i = (lo + hi) / 2;
  if (a [i] == x) return i;
  if (a [i] < x) lo = i+1; // 在 a [i+1..hi] 中连续查找
  else hi = i-1;         // 在 a [lo..i-1] 中连续查找
}
return n; // 在 a [0.. n-1] 中没找到 x
}
index (44, a, 7); // 2

```

这里，数组 `a []` 没有完全排序，但是它的头 7 个元素是有序的，所以在第一次调用 `index (44, a, 7)` 时，`index()` 函数调用 `isNondecreasing (a, 7)`，将布尔值 `true` 返回给 `assert ()` 函数，且输出与例 6.14 相同。但在第二次调用 `index (44, a, 8)` 时，调用 `isNondecreasing (a, 8)` 将布尔型值 `false` 返回给 `assert ()` 函数，将终止该程序，使得 Windows 显示如图 6.10 所示的警告框。

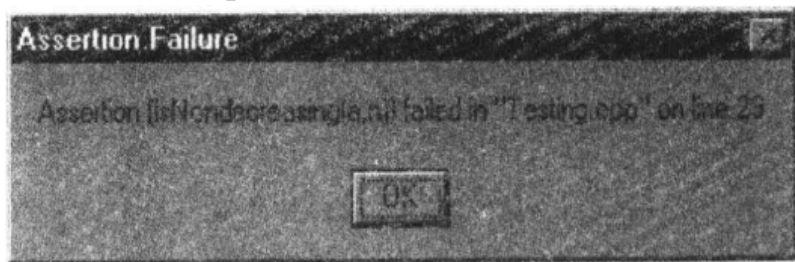


图 6.10 错误指示

## 6.9 使用枚举类型的数组

枚举类型是在第 2 章中介绍的，很自然地可以采用数组来处理枚举类型

### 例 6.17 枚举一星期中的每一天

下例定义一个含有 7 个 `float` 型元素的数组 `high []`，表示一星期中 7 天的最高温度：

```

int main ()
{ enum Day { SUN, MON, TUE, WED, THU, FRI, SAT };
  float high [SAT+1] = {88.3, 95.0, 91.2, 89.9, 91.4, 92.5, 89.7};
  for (int day = SUN; day <= SAT; day++)
    cout << "The high temperature for day " << day
          << " was " << high [day] << endl;
}

```

The high temperature for day 0 was 88.3  
 The high temperature for day 1 was 95.0  
 The high temperature for day 2 was 91.2  
 The high temperature for day 3 was 89.9

```
The high temperature for day 4 was 91.4
The high temperature for day 5 was 92.5
The high temperature for day 6 was 86.7
```

该数组的大小是 SAT + 1, 因为 SAT 的值为整数 6, 而数组需要 7 个元素。

在 for 循环中作为一个下标定义的 int 型变量 day 取值为 SUN、MON、TUE、WED、THU、FRI 或 SAT。记住这些值实际上是作为整数 0、1、2、3、4、5 和 6 存放的。

注意不可能打印出符号常量的名字。

用这种方法使用枚举常量的好处是使得程序代码“自动说明”。例如, 在例 6.17 中, for 循环控制

```
for (int day = SUN; day <= SAT; day++)
```

代表了它自己。

## 6.10 类型定义

枚举类型是程序员用来定义他们自己的数据类型的方法之一。例如,

```
enum Color {RED, ORANGE, YELLOW, GREEN, BLUE, VIOLET};
```

定义了 Color 类型, 可以用来声明变量, 如下所示:

```
Color shirt = BLUE;
Color car [] = {GREEN, RED, BLUE, RED};
Float wavelength [VIOLET+1] = {420, 480, 530, 570, 600, 620};
```

这里, shirt 是一个变量, 可以在 Color 类型的 6 个值中任取一个值, 并被初始化为 BLUE, car 是一个含有 4 个 Color 类型元素的数组, 下标范围从 0 到 3, wavelength 是一个含有 6 个 float 类型元素的数组, 下标范围从 RED 到 VIOLET。

C++ 还提供一种重命名已存在类型的方法, 关键字 typedef 声明了一个特定类型的新名字 (即一个同义字或别名)。语法为:

```
typedef 类型 别名;
```

其中, 类型是给定的类型, 别名是新名字。如果已习惯使用 Pascal 编程, 可能想使用如下的类型别名:

```
typedef long Integer;
typedef double Real;
```

则可以这样来声明变量:

```
Integer n = 22;
Const Real PI = 3.141592653589793;
Integer frequency [64];
```

注意，重命名一个数组类型的 typedef 语句的语法为：

**typedef** 元素类型 别名 [];

这表示数组元素的个数不是数组类型的一部分。

一个 typedef 语句不是定义一个新的类型；它仅是为一个已存在的类型提供一个同义字。例如，在例 6.9 中定义的 sum () 函数可以这样调用：

```
cout << sum (frequency, 4);
```

虽然上面声明数组 frequency [] 时数组元素的类型为 Integer，因为对于同一类型来说 Integer 和 int 是同一个名字，所以在参数之间是没有冲突的。

下例说明了 typedef 的另一个用法。

### 例 6.18 另一个冒泡排序的例子

该例与例 6.13 的运行结果相同，唯一的改变是使用 typedef 来重命名的 Sequence 类型，该类型将在 main () 中用于参数列表和 a 的声明：

```
typedef float Sequence [];  
void sort (Sequence, int);  
void print (Sequence, int);  
int main ()  
{ Sequence a = {55.5, 22.5, 99.9, 66.6, 44.4, 88.8, 33.3, 77.7};  
  print (a, 8);  
  sort (a, 8);  
  print (a, 8);  
}  
void sort (Sequence a, int n)  
{ for (int i = n - 1; i > 0; i -- )  
  for (int j = 0; j < i; j ++ )  
    if (a [j] > a [j + 1]) swap (a [j], a [j + 1]);  
}
```

注意 typedef 语句：

```
typedef float Sequence [];
```

括号 [] 是在给类型取别名时的 Sequence 后面出现的，然后当该别名用于声明数组变量和形式参数时就不带括号了。

## 6.11 多维数组

前面所使用的数组都是一维的，这表示它们是线性的，即有顺序的。但是一个数组的元素类型几乎可以是任意类型，包括数组类型。元素为数组的数组称为多维数组，元素为一维数组的一维数组称为二维数组；元素为二维数组的一维数组是三维数组，依次类推。

声明一个多维数组的最简单的方法如下所示：

```
double a [32] [10] [4];
```

这是一个每维大小分别为 32, 10, 4 的三维数组 语句

```
a [25] [8] [3] = 99.99;
```

将会给通过多下标 (25,8,3) 指定的元素赋值为 99.99。

### 例 6.19 读入和打印一个二维数组

本例说明如何处理一个二维数组：

```
void read (int a [] [5]);
void print (const int a [] [5]);

int main ()
{ int a [3] [5];
  read (a);
  print (a);
}

void read (int a [] [5])
{ cout << "Enter 15 integers, 5 per row: \n";
  for (int i=0; i<3; i++)
  { cout << "Row" << i << ": ";
    for (int j=0; j<5; j++)
      cin >> a [i] [j];
  }
}

void print (const int a [] [5])
{ for (int i=0; i<3; i++)
  { for (int j=0; j<5; j++)
    cout << " " << a [i] [j];
    cout << endl;
  }
}
```

```
Enter 15 integers, 5 per row:
Row 0: 44 77 33 11 44
Row 1: 60 50 30 90 70
Row 2: 85 25 45 45 55
44 77 33 11 44
60 50 30 90 70
85 25 45 45 55
```

注意在函数的参数列表中，没有指定一维而指定了二维 (5)，这是因为二维数组 `a[] []` 是作为三个 5 元素数组的一维数组存放的。编译器不需要知道有多少个这样的 5 元素数组要存放，但是它需要知道它们是 5 元素的数组。

当一个多维数组被传递给一个函数时，不指定第一维，而其余的维是要指定的。

### 例 6.20 处理一个存放测验分数的二维数组

```
const NUM_STUDENTS = 3;
```



```

const NUM_QUIZZES = 5;
typedef int Score [NUM_STUDENTS][NUM_QUIZZES];
void read (Score);
void printQuizAverages (Score);
void printClassAverages (Score);
int main ()
{ Score score;
  cout << "Enter " << NUM_QUIZZES << " scores for each student: \n";
  read (score);
  cout << "The quiz averages are: \n";
  printQuizAverages (score);
  cout << "The class averages are: \n";
  printClassAverages (score);
}

void read (Score score)
{ for (int s=0; s<NUM_STUDENTS; s++)
  { cout << "Student " << s << ": ";
    for (int q=0; q<NUM_QUIZZES; q++)
      cin >> score [s] [q];
  }
}

void printQuizAverages (Score score)
{ for (int s=0; s<NUM_STUDENTS; s++)
  { float sum = 0.0;
    for (int q=0; q<NUM_QUIZZES; q++)
      sum += score [s] [q];
    cout << "\tStudent " << s << ": " << sum/NUM_QUIZZES << endl;
  }
}

void printClassAverages (Score score)
{ for (int q=0; q<NUM_QUIZZES; q++)
  { float sum = 0.0;
    for (int s=0; s<NUM_STUDENTS; s++)
      sum += score [s] [q];
    cout << "\tQuiz " << q << ": " << sum/NUM_STUDENTS << endl;
  }
}

```

Enter 5 quiz scores for each student:

Student 0: 8 7 9 8 9

Student 1: 9 9 9 9 8

Student 2: 5 6 7 8 9

The quiz averages are:

Student 0: 8.2

Student 1: 8.8

Student 2: 7

The class averages are:

Quiz 0: 7.33333

Quiz 1: 7.33333

Quiz 2: 8.33333

Quiz 3: 8.33333

Quiz 4: 8.66667

这里使用了一个 typedef 来定义二维数组类型的别名,使得函数头的可读性更好。

printQuizAverages() 函数打印每 3 行分数的平均值,而 printClassAverages() 函数打印每 5 列分数的平均值。

### 例 6.21 处理一个三维数组

本例简单计算一个三维数组中的 0 的个数:

```
int numZeros (int a [][4][3], int n1, int n2, int n3);
int main ()
{ int a [2][4][3] = { { {5, 0, 2}, {0, 0, 9}, {4, 1, 0}, {7, 7, 7} },
                      { {3, 0, 0}, {8, 5, 0}, {0, 0, 0}, {2, 0, 9} }
                    };
  cout << "This array has " << numZeros (a, 2, 4, 3) << " zeros: \n";
}
int numZeros (int a [][4][3], int n1, int n2, int n3)
{ int count = 0;
  for (int i = 0; i < n1; i++)
    for (int j = 0; j < n2; j++)
      for (int k = 0; k < n3; k++)
        if (a [i] [j] [k] == 0) ++count;
  return count;
}
```

This array has 11 zeros:

注意数组是如何初始化的:它是一个有 3 个元素的 4 元素数组的 2 元素数组,总共有 24 个数组元素,可以这样初始化:

```
int a [2][4][3] = {5,0,2,0,0,9,4,1,0,7,7,7,3,0,0,8,5,0,0,0,0,2,0,9};
```

或这样初始化:

```
int a [2][4][3] = {{5,0,2,0,0,9,4,1,0,7,7,7},{3,0,0,8,5,0,0,0,0,2,0,9}};
```

但是这两种形式都比使用三维初始化表更加难以阅读和理解。

还要注意三个嵌套的 for 循环,使用了 d 层 for 循环处理一个 d 维数组,每维用一层循环处理。

## 复 习 题

- 6.1 数组中的元素可以有多少种不同的数据类型?
- 6.2 数组下标必须具有什么样的类型和范围?
- 6.3 当一个数组在声明时未包含初始化表时,其元素的值将是什么?
- 6.4 当声明一个数组时,初始化表中的值少于数组元素的个数,其元素的值将是什么?
- 6.5 当数组的初始化表中的值的个数多于数组的大小时,将会发生什么现象?
- 6.6 如何区分 enum 语句和 typedef 语句?

- 6.7 当一个多维数组被传递给一个函数时,为什么C++需要在参数表中指定除一维以外的所有信息?

## 习 题

- 6.1 修改例6.1中的程序,使得它可以提示每个输入,并可标出每个输出,如下所示:

```
Enter 5 numbers
a [0]: 11.11
a [1]: 33.33
a [2]: 55.55
a [3]: 77.77
a [4]: 99.99
In reverse order, they are:
a [4] = 99.99
a [3] = 77.77
a [2] = 55.55
a [1] = 33.33
a [0] = 11.11
```

- 6.2 修改例6.1中的程序,使得它可以将数组倒序存放,然后按存放顺序打印该数组,如下所示:

```
Enter 5 numbers
a [4]: 55.55
a [3]: 66.66
a [2]: 77.77
a [1]: 88.88
a [0]: 99.99
In reverse order, they are:
a [0] = 99.99
a [1] = 88.88
a [2] = 77.77
a [3] = 66.66
a [4] = 55.55
```

- 6.3 修改例6.9中的程序,使得它可以测试以下的函数:

```
float ave (int [] a, int n);
//返回 a [] 的前 n 个元素的平均值
```

- 6.4 修改例6.10中的程序,使得它可以打印数组、数组元素的和及数组元素的平均值(参见例6.9和习题6.3)。
- 6.5 修改例6.11中的程序,使得它可以打印每个数组元素的内存地址和单元内容。对于一个名为a的数组,使用表达式a, a+1, a+2等,来存放元素a[0], a[1], a[2]等的地址,并且使用表达式\*a, \*(a+1), \*(a+2)等来存放这些地址的内容,将数组声明为:

```
unsigned int n);
```

使得当将数组元素插入到 `cout` 流中时, 可以作为整数打印出来

6.6 修改例 6.12 中的程序, 使得它不返回第一个元素的地址, 而是返回最后一个元素的地址

6.7 修改例 6.15 中的程序, 当且仅当数组不是递增序时, 该函数返回 `true`。

6.8 写出并测试以下函数, 返回给定数组中的前  $n$  个元素的最小值。

```
float min (float a [], int n);
```

6.9 写出并测试以下函数, 返回给定数组中的前  $n$  个元素中最小值的下标。

```
int minIndex (float a [], int n);
```

6.10 写出并测试以下函数, 通过引用参数返回数组中的最大值和最小值。

```
void getExtremes (float& min, float& max, float a [], int n);
```

6.11 写出并测试以下函数, 通过引用参数返回数组中的最大值和第二大值 (二者可能相等)。

```
void largest (float& max1, float& max2, float a [], int n);
```

6.12 写出并测试以下函数, 从数组中删除一个元素:

```
void remove (float a [], int& n, int i);
```

该函数通过将 `a[i]` 后面的  $n$  个元素依次向前移动一位来删除 `a[i]`。

6.13 写出并测试以下函数, 从数组中删除一个元素:

```
bool removeFirst (float a [], int& n, float x);
```

该函数在数组的前  $n$  个元素中查找  $x$ 。如果找到  $x$ , 则删除找到的第一个  $x$ , 该位置后面的所有元素依次向前移动一位,  $n$  减 1, 并返回 `true` 表示删除成功。如果未找到  $x$ , 则数组不变, 并返回 `false` (参见习题 6.12)。

6.14 写出并测试以下函数, 从数组中删除元素:

```
bool removeAll (float a [], int& n, float x);
```

该函数从数组 `a` 的前  $n$  个元素中删除所有的  $x$ , 并且从  $n$  中减去所删除的  $x$  的个数 (参见习题 6.13)。

6.15 写出并测试以下函数:

```
void rotate (int a [], int n, int k);
```

该函数“旋转”数组 `a` 的前  $n$  个元素, 将其右边的  $k$  个元素 (如果  $k$  为负数, 则取左边的  $-k$  个元素) 移动到数组的开始位置。例如, 调用 `rotate(a, 8, 3)` 将数组 `{22, 33, 44, 55, 66, 77, 88, 99}` 变为 `{77, 88, 99, 22, 33, 44, 55, 66}`, 而

调用 `rotate (a, 8, -5)` 的结果与上面相同。

6.16 写出并测试以下函数：

```
void append (int a [], int m, int b [], int n);
```

该函数将数组 `b` 的前 `n` 个元素追加到数组 `a` 的前 `m` 个元素之后。假定数组 `a` 具有至少存放 `m + n` 个元素的空间。例如，如果 `a` 为 `{22, 33, 44, 55, 66, 77, 88, 99}` 且 `b` 为 `{20, 30, 40, 50, 60, 70, 80, 90}`，则调用 `append (a, 5, b, 3)` 将把 `a` 变为 `{22, 33, 44, 55, 66, 20, 30, 40}`。注意 `b` 并没有改变，且 `a` 中只改变了 `n` 个元素。

6.17 写出并测试以下函数：

```
void insert (float a [], int& n, float x);
```

该函数将值 `x` 插入有 `n` 个元素的有序数组 `a` 中，然后 `n` 加 1。插入 `x` 后要保持数组依然有序，这要求移动一些元素来为新插入的 `x` 空出位置。（注意，要求数组至少能存放 `n + 1` 个元素）

6.18 实现插入排序算法，对有 `n` 个元素的数组 `a` 排序。在该算法中，主循环控制变量 `i` 从 1 到 `n - 1`。在第 `i` 次循环中，将元素 `a [i]` 按正确位置插入到子数组 `a [0..i]` 中，方法为：首先将该子数组中所有大于 `a [i]` 的元素向后移动一个位置，然后将 `a [i]` 复制到小于或等于 `a [i]` 的元素与大于 `a [i]` 的元素中间的位置。（提示：使用习题 6.17 中的 `insert ()` 算法）

6.19 实现选择排序算法，对有 `n` 个元素的数组 `a` 排序。在该算法中使用 `n - 1` 次循环，每次循环选择下一个最大的元素 `a [j]`，并将它与其应在的正确位置上的元素交换位置。所以第一次循环在所有的元素中选出最大的元素并与 `a [n - 1]` 交换位置，第二次循环在剩余的未排序的数组 `a [0..n - 2]` 中选出最大的元素并与 `a [n - 2]` 交换位置，依此类推，第 `i` 次循环在剩余的未排序的数组 `a [0..n - i]` 中选出最大的元素并与 `a [n - i]` 交换位置。（提示：使用与例 6.13 中相同的循环）

6.20 改写并测试例 6.13 中的冒泡排序函数，变为间接排序算法，通过将数组下标排序实现数组有序，而不是移动实际的数组元素。

6.21 写出并测试函数：

```
int frequency (float a [], int n, int x);
```

该函数计算数组 `a` 的前 `n` 个元素中值 `x` 出现的次数，然后返回该次数值。

6.22 实现埃拉托色尼之筛来查找素数。定义一个名为 `isPrime [SIZE]` 的布尔型数组，设置 `isPrime [0]` 和 `isPrime [1]` 的值为 `false`（2 是第一个素数），并设置其余的元素为 `true`，然后对从 4 到 `SIZE - 1` 的每个 `i`，如果 `i` 能被 2 整除（即 `i % 2 == 0`）则设置 `isPrime [i]` 为 `false`，再对从 6 到 `SIZE - 1` 的每个 `i`，如果 `i` 能被 3 整除则设置 `isPrime [i]` 为 `false`。对从 2 到 `SIZE / 2` 的每个可能的因子重复以上的处理，当操作结束时，所有值仍为 `true` 的 `is-`

Prime[i] 所对应的 i 就是素数, 它们是从筛子中掉下来的数。

6.23 写出并测试以下的函数:

```
void reverse (int a [], int n);
```

该函数颠倒数组的前 n 个元素的顺序。例如: 调用 reverse (a, 5) 将会把数组 {22, 33, 44, 55, 66, 77, 88, 99} 变为 {66, 55, 44, 33, 22, 77, 88, 99}。

6.24 写出并测试以下的函数:

```
bool isSymmetric (int a [], int n);
```

当且仅当将数组的前 n 个元素颠倒次序而数组不变时, 该函数返回 true。例如, 如果 a 为 {22, 33, 44, 55, 44, 33, 22}, 则调用 isSymmetric (a, 7) 将返回 true, 但是调用 isSymmetric (a, 4) 将返回 false。警告: 该函数将不会改变该数组。

6.25 写出并测试以下的函数:

```
void add (float a [], int n, float b []);
```

该函数将数组 b 的前 n 个元素加到数组 a 的相应的前 n 个元素上。例如, 如果数组 a 为 {2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9} 且 b 为 {6.0, 5.0, 4.0, 3.0, 2.0, 1.0}, 则调用 add (a, 5, b) 将把 a 变为 {8.2, 8.3, 8.4, 8.5, 8.6, 7.7, 8.8, 9.9}。

6.26 写出并测试以下的函数:

```
void multiply (float a [], int n, float b []);
```

该函数将数组 a 的前 n 个元素与数组 b 的相应的前 n 个元素相乘。例如, 如果数组 a 为 {2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9} 且 b 为 {4.0, -3.0, 2.0, -1.0, 0.0, 0.0}, 则调用 multiply (a, 5, b) 将把 a 变为 {8.8, -9.9, 8.8, -5.5, 0.0, 7.7, 8.8, 9.9}。

6.27 写出并测试以下的函数:

```
float innerProduct (float a [], int n, float b []);
```

该函数返回数组 a 的前 n 个元素与数组 b 的相应的前 n 个元素的内积 (也称为“点积”或“矢量积”), 其定义为相应元素乘积的和。例如, 如果数组 a 为 {2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9} 且 b 为 {4.0, -3.0, 2.0, -1.0, 0.0, 0.0}, 则调用 innerProduct (a, 5, b) 将返回  $(2.2)(4.0) + (3.3)(-3.0) + (2.2)(4.0) + (5.5)(-1.0) + (6.6)(0.0) = 2.2$ 。

6.28 写出并测试以下的函数:

```
float outerProduct3 (float p [] [3], float a [], float b []);
```

该函数返回数组 a 的前 3 个元素与数组 b 的相应的前 3 个元素的外积。例如, 如果数组

a 为 {2.2, 3.3, 4.4} 且 b 为 {2.0, -1.0, 0.0}, 则调用 `outerProduct3 (p, a, b)` 将把二维数组 p 变为:

4.4 -2.2 0.0

6.6 -3.3 0.0

8.8 -4.4 0.0

6.29 写出并测试一个函数, 完全打乱一个元素个数为奇数的一维数组。例如, 它可以将数组 {11, 22, 33, 44, 55, 66, 77, 88} 变为 {11, 55, 22, 66, 33, 77, 44, 88}。

6.30 写出并测试一个函数, 将一个二维数组顺时针旋转 90°。例如, 它将数组

11 22 33

44 55 66

77 88 99

变为:

77 44 11

88 55 22

99 66 33

6.31 写出并运行一个程序, 该程序读入一些未指明的数, 然后打印这些数及它们与其平均值的偏差

6.32 写出并测试以下函数:

```
double stdev (double x [], int n);
```

该函数返回一个含有  $n$  个数字的数据序列  $x_0, \dots, x_{n-1}$  的标准偏差, 标准偏差的定义式为:

$$s = \sqrt{\frac{\sum_{i=0}^{n-1} (x_i - \bar{x})^2}{n - 1}}$$

其中  $\bar{x}$  是数据的均值。该公式为: 首先求每个偏差  $(x[i] - \text{mean})$  的平方; 然后将这些平方数相加; 再除以  $n - 1$ ; 最后求其平方根。

6.33 改进问题 6.31 中的程序, 使它也可以计算输入数据的 Z 值。 $n$  个数  $x_0, \dots, x_{n-1}$  的 Z 值定义为:  $z_i = (x_i - \bar{x}) / s$ , 它们将给定的数规格化, 使这些数以 0.0 为中心, 且平均偏差为 1.0。使用问题 6.32 中的函数。

6.34 在假想的“good old days”中, 当等级“C”被认为是“平均的”, 大班的教师经常会将班里的分数根据以下的公式统计成曲线图表:

A:  $1.5 \leq z$

B:  $0.5 \leq z < 1.5$

C:  $-0.5 \leq z < 0.5$

D:  $-1.5 \leq z < -0.5$

E:  $z < -1.5$

如果等级是正态分布的（即它们的密度曲线是钟状分布的），则该算法的结果为 7% A, 24% B, 38% C, 24% D, 7% E。这里的  $z$  值就是问题 6.33 中的  $Z$  值。改进问题 6.33 中的程序，使它对所有读入的测验分数，可以打印出该“曲线”。

- 6.35 写出并测试一个函数，该函数在传递给它的正方形矩阵中生成 Pascal 三角形。例如，如果传递二维数组  $a$  和整数 4 给该函数，则它会将  $a$  赋值为：

```
1 0 0 0 0
1 1 0 0 0
1 2 1 0 0
1 3 3 1 0
1 4 6 4 1
```

- 6.36 按照 John von Neumann 的游戏和经济行为的理论，某个两人游戏可以由一个称为盈利矩阵的二维数组表示。当盈利矩阵具有一个鞍点时，玩游戏者就可以获得最佳策略。鞍点是矩阵中同时是 minimax 和 maximin 的数。一个矩阵的 minimax 指的是最大的列中的最小值，maximin 指的是最小的行中的最大值，当两个值相等时就可能获得最佳策略。写出一个程序，可以打印一个给定矩阵的 minimax 和 maximin。

## 复习题答案

- 6.1 只有一种：一个数组中的元素必须具有相同的数据类型。
- 6.2 数组下标必须是整数类型，范围为从 0 到  $n-1$ ，其中  $n$  为数组的大小。
- 6.3 如果没有初始化表，数组元素将具有不可预测的初值。
- 6.4 如果一个数组的初始化表中的值少于数组元素的个数，则这些值将会赋给数组前面的元素，而剩余的元素将会自动初始化为 0。
- 6.5 当数组的初始化表中的值的个数多于数组的大小时，将会出错。
- 6.6 enum 语句用来定义枚举类型，这是一种新的 unsigned 整数类型，而 typedef 语句仅仅定义一个已存在类型的别名。
- 6.7 当一个多维数组被传递给一个函数时，要指定除一维以外的所有维，以便编译器可以计算数组的每个元素的位置。

## 习题答案

- 6.1 修改例 6.1 中的程序，增加输入提示和输出标签：

```
int main()
{ const int SIZE=5;
  double a[SIZE];
  cout << "Enter " << SIZE << " numbers:\n";
```



```

    for (int i=0; i<SIZE; i++)
    { cout << "\ta[" << i << "]: ";
      cin >> a[i];
    }
    cout << "In reverse order, they are:\n";
    for (int i=SIZE-1; i>=0; i--)
        cout << "\ta[" << i << "] = " << a[i] << endl;
}

```

6.2 修改例 6.1 中的程序,使得输入可以按倒序存放:

```

int main()
{ const int SIZE=5;
  double a[SIZE];
  cout << "Enter " << SIZE << " numbers:\n";
  for (int i=SIZE-1; i>=0; i--)
  { cout << "\ta[" << i << "]: ";
    cin >> a[i];
  }
  cout << "In reverse order, they are:\n";
  for (int i=0; i<SIZE; i++)
      cout << "\ta[" << i << "] = " << a[i] << endl;
}

```

6.3 修改例 6.9 中的程序,使得它可以测试返回数组元素平均值的函数:

```

float ave(int[],int);
int main()
{ int a[] = { 11, 33, 55, 77 };
  int size = sizeof(a)/sizeof(int);
  cout << "ave(a,size) = " << ave(a,size) << endl;
}
float ave(int a[], int n)
{ float sum=0.0;
  for (int i=0; i<n; i++)
      sum += a[i];
  return sum/n;
}

```

6.4 修改例 6.10 中的程序,使得它可以打印数组、数组元素的和及数组元素的平均值:

```

void read(int[],int&);
void print(int[],int);
int sum(int[],int);
float ave(int[],int);
int main()
{ const int MAXSIZE=100;
  int a[MAXSIZE]={0}, size;
  read(a,size);
  cout << "The array has " << size << " elements: ";
  print(a,size);
  cout << "\nIts sum is " << sum(a,size)
      << "\nand its average is " << ave(a,size) << endl;
}

```

函数定义与例 6.9、例 6.10 和问题 6.3 中相同。

### 6.5 修改例 6.11 中的程序，使得它可以打印每个数组元素的内存地址和单元内容：

```
int main()
{ unsigned int a[] = { 22, 44, 66, 88 };
  cout << "a   = " << a   << ", *a   = " << *a   << endl;
  cout << "a+1 = " << a+1 << ", *(a+1) = " << *(a+1) << endl;
  cout << "a+2 = " << a+2 << ", *(a+2) = " << *(a+2) << endl;
  cout << "a+3 = " << a+3 << ", *(a+3) = " << *(a+3) << endl;
}
```

```
a   = 0x0064fdbc, *a   = 22
a+1 = 0x0064fdb0, *(a+1) = 44
a+2 = 0x0064fdb4, *(a+2) = 66
a+3 = 0x0064fdb8, *(a+3) = 88
```

在每个内存地址前面的 0x 表示地址是十六进制数（许多计算机都是用十六进制符号来表示地址的）。注意每个地址在它前面一个地址的下面 4 个字节，这表示 unsigned int 型的对象在内存中占用 4 字节。

### 6.6 修改例 6.12 中的程序，使得它不返回第一个元素的地址，而是返回最后一个元素的地址：

```
int index(int, int[], int);
int main()
{ int a[] = { 22, 44, 66, 88, 44, 66, 55 };
  cout << "index(44,a,7) = " << index(44,a,7) << endl;
  cout << "index(50,a,7) = " << index(50,a,7) << endl;
}
int index(int x, int a[], int n)
{ for (int i=n-1; i>=0; i--)
  if (a[i] == x) return i;
  return n;
}
```

```
index (44, a, 7) = 4
index (50, a, 7) = 7
```

### 6.7 修改例 6.15 中的程序，使得它可以判断数组是否为非递增序：

```
bool isNonincreasing(int a[], int n)
{ for (int i=1; i<n; i++)
  if (a[i]>a[i-1]) return false;
  return true;
}
```

### 6.8 float min (float a [] int n)

```
{ assert(n >= 0);
  float min=a[0];
  for (int i=1; i<n; i++)
    if (a[i] < min) min = a[i];
  return min;
}
```

6.9 int minIndex (float a [], int n)

```
{ assert(n >= 0);
  int j=0;
  for (int i=1; i<n; i++)
    if (a[i] < a[j]) j = i;
  return j;
}
```

6.10 void getExtremes (float& min, float& max, float a [], int n)

```
{ assert(n >= 0);
  min = max = a[0];
  for (int i=1; i<n; i++)
    if (a[i] < min) min = a[i];
    else if (a[i] > max) max = a[i];
}
```

6.11 void largest (float& max2, float a [], int n)

```
{ assert(n >= 1);
  if (n == 1) return a[0];
  int i1=0, i2;
  for (int i=1; i<n; i++)
    if (a[i] > a[i1]) i1 = i;
  max1 = a[i1];
  i2 = ( i1 == 0 ? 1 : 0 );
  for (int i=i2+1; i<n; i++)
    if (i != i1 && a[i] > a[i2]) i2 = i;
  max2 = a[i2];
}
```

6.12 void remove (float a [], int& n, int i)

```
{ for (int j=i+1; j<n; j++)
  a[j-1] = a[j];
  --n;
}
```

6.13 bool removeFirst (float a [], int& n, float x)

```
{ for (int i=0; i<n; i++)
  if (a[i] == x)
  { for (int j=i+1; j<n; j++)
    a[j-1] = a[j];
    --n;
    return true;
  }
  return false;
}
```

6.14 void removeAll (float a [], int& n, float x)

```

{ for (int i=0; i<n; i++)
    if (a[i] == x)
    { for (int j=i+1; j<n; j++)
        a[j-1] = a[j];
        --n;
    }
}

```

#### 6.15 void rotate (int a [], int n, int k)

```

{ const int MAXOFFSET=100;
  assert(k < MAXOFFSET);
  int temp[MAXOFFSET];
  if (k > 0)
  { for (int j=0; j<k; j++) // 把 k 个元素复制到 temp[] 中
      temp[j] = a[n-k+j];
    for (int i=n-1; i>=k; i--) // 移动 n-k 个元素
      a[i] = a[i-k];
    for (int i=0; i<k; i++) // 把 k 个元素复制回到 a[] 中
      a[i] = temp[i];
  }
  if (k < 0)
  { for (int j=0; j<-k; j++) // 把 k 个元素复制到 temp[] 中
      temp[j] = a[j];
    for (int i=0; i<n+k; i++) // 移动 n+k 个元素
      a[i] = a[i-k];
    for (int i=n+k; i<n; i++) // 把 k 个元素复制回到 a[] 中
      a[i] = temp[i-n-k];
  }
}

```

#### 6.16 void append (int a [], int m, int b [], int n)

```

{ for (int j=0; j<n; j++) // 把 n 个元素复制到 a[] 中
    a[m+j] = b[j];
}

```

#### 6.17 void insert (float a [], int& n, float x)

```

{ int j=n;
  while (j>0 && a[j-1]>x)
    a[j--] = a[j-1];
  a[j] = x;
  ++n;
}

```

#### 6.18 void sort (float a [], int n)

```

{ // 插入排序
  for (int i=1; i<n; i++)
  { // 把 a[i] 插入到 a[0..i-1] 中
    float x=a[i];

```

```

    int j=i;
    while (j>0 && a[j-1]>x)
        a[j--] = a[j-1];
    a[j] = x;
    // 不变量: a[0..i] 是排序的
}
}

```

#### 6.19 void sort (float a [], int n)

```

{ // selection sort:
  for (int i=1; i<n; i++)
  { // 选择 a[k]=max{a[0],a[1],...,a[n-1]}
    int k=0;
    for (int j=1; j<=n-i; j++)
      if (a[j]>a[k]) k = j;
    swap(a[k],a[n-i]);
    //不变量: a[n-1-i..n-1] 是排序的
  }
}

```

#### 6.20 void sort (float a [], int indx [], int n)

```

{ // 间接的冒泡排序
  for (int i=1; i<n; i++)
    //冒泡排序的最大值 {a[0],a[1],...,a[n-1]}
    for (int j=0; j<n-i; j++)
      if (a[indx[j]] > a[indx[j+1]]) swap(indx[j],indx[j+1]);
    // 不变量: a[indx[n-1-i]]<=a[indx[n-i]]<=..a[indx[n-1]]
}

```

#### 6.21 int frequency (float [], int, int);

```

int main()
{ float a[] = {561, 508, 400, 301, 329, 599, 455, 400, 346, 346,
               329, 375, 561, 390, 399, 400, 401, 561, 405, 405,
               455, 508, 473, 329, 561, 505, 329, 455, 561, 599,
               561, 455, 346, 301, 455, 561, 399, 599, 508, 508};

  int n=40, x;
  cout << "Item: ";
  cin >> x;
  cout << x << " has frequency " << frequency(a,n,x) << endl;
}

int frequency(float a[], int n, int x)
{ int count = 0;
  for (int i=0; i<n; i++)
    if (a[i] == x) ++count;
  return count;
}

```

Item: 400

400 has frequency 3

```

6.22 #include <iomanip> // 定义 setw() 函数
#include <iomanip> // 定义 setw() 函数
#include <iostream> // 定义 cout() 对象
using namespace std;
const int SIZE = 400;
void sieve(bool[], int);
void print(bool[], int);
int main()
{ // 打印所有小于 SIZE 的素数
    bool isPrime[SIZE] = {0};
    sieve(isPrime, SIZE);
    print(isPrime, SIZE);
}
void sieve(bool isPrime[], int n)
{ // sets isPrime[i] = false iff i is not prime:
    for (int i=2; i<n; i++)
        isPrime[i] = true; // 假设所有 i 大于 1 的数都是素数
    for (int p=2; p<=n/2; p++)
        for (int m=2*p; m<n; m += p)
            isPrime[m] = false; // p 的倍数不是素数
}
void print(bool a[], int n)
{ // 当 isPrime[i] 为真时, 打印每个 i
    for (int i=1; i<n; i++)
        if (a[i]) cout << setw(3) << i;
        else cout << setw(3) << (i%20==0 ? '\n' : ' ');
}

```

	2	3	5	7		11	13		17	19
		23			29	31		37		
41		43		47			53			59
61				67		71	73			79
		83			89			97		
101	103		107	109			113			
			127		131			137	139	
				149	151			157		
	163		167			173			179	
181					191	193		197	199	
					211					
	223		227	229		233			239	
241					251			257		
	263			269	271			277		
281	283					293				
			307		311	313		317		
					331			337		
			347	349		353			359	
			367			373			379	
	383			389				397		

```

6.23 void reverse (int a [], int n)
{ for (int i=0; i<n/2; i++)
    swap(a[i], a[n-1-i]);
}

```

- ```

6.24 bool issymmetric (int a [], int n)
{ for (int i=0; i<n/2; i++)
    if (a[i] != a[n-1-i]) return false;
    return true;
}

6.25 void add (float a [], int n, float b [])
{ for (int i=0; i<n; i++)
    a[i] += b[i];
}

6.26 void multiply (float a [], int n, float b [])
{ for (int i=0; i<n; i++)
    a[i] *= b[i];
}

6.27 float innerProduct (float a [], int n, float b [])
{ float p=0;
  for (int i=0; i<n; i++)
    p += a[i]*b[i];
  return p;
}

6.28 void outerProduct3 (float p [] [3], float a [], float b [])
{ for (int i=0; i<3; i++)
    for (int j=0; j<3; j++)
        p[i][j] = a[i]*b[j];
}

6.29 void shuffle (int a [], int n)
{ // 对偶数个元素的完全打乱法
  assert(n <= SIZE);
  int temp[SIZE];
  for (int i=0; i<n/2; i++)
  { temp[2*i] = a[i];
    temp[2*i+1] = a[n/2+i];
  }
  for (int i=0; i<n; i++)
    a[i] = temp[i];
}

6.30 const int SIZE=3;
typedef int Matrix[SIZE][SIZE];
void print(Matrix);
void rotate(Matrix);
int main()
{ // 测试 rotate() 函数
  Matrix m = { 11, 22, 33, 44, 55, 66, 77, 88, 99 };
  print(m);
  rotate(m);
}

```

```

    print(m);
}
void print(Matrix a)
{ for (int i=0; i<SIZE; i++)
  { for (int j=0; j<SIZE; j++)
    cout << a[i][j] << "\t";
    cout << endl;
  }
  cout << endl;
}
void rotate(Matrix m)
{ Matrix temp;
  for (int i=0; i<SIZE; i++)
    for (int j=0; j<SIZE; j++)
      temp[i][j] = m[SIZE-j-1][i];
  for (int i=0; i<SIZE; i++)
    for (int j=0; j<SIZE; j++)

      m[i][j] = temp[i][j];
}

```

6.31

```

const int SIZE = 100;
void read(double[], int&);
double mean(double[], int);
int main()
{ double x[SIZE];
  int n=0;
  read(x, n);
  double m = mean(x, n);
  cout << "mean = " << m << endl;
  for (int i = 0; i < n; i++)
    cout << "x[" << i << "] = " << x[i]
      << ", dev[i] = " << x[i] - m << endl;
}
void read(double x[], int& n)
{ cout << "Enter data. Terminate with 0:\n";
  while (n<SIZE)
  { cout << "x[" << n << "]: ";
    cin >> x[n];
    if (x[n] == 0) break;
    else ++n;
  }
}
double mean(double x[], int n)
{ double sum=0;
  for (int i=0; i<n; i++)
    sum += x[i];
  return sum/n;
}

```



```

Enter data. Terminate with 0:
x[0]: 1.23
x[1]: 7.65
x[2]: 0
mean = 4.44
x[0] = 1.23, dev[i] = -3.21
x[1] = 7.65, dev[i] = 3.21

```

- ```

6.32 double stdev(double a[], int n)
{ assert(n > 1);
  double sum=0;
  for (int i=0; i<n; i++)
    sum += a[i];
  double mean = sum/n;
  sum=0;
  double deviation;
  for (int i=0; i<n; i++)
  { deviation = a[i] - mean;
    sum += deviation*deviation;
  }
  return sqrt(sum/(n-1));
}

6.33 int main()
{ double x[] = { 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9 };
  int n=8;
  print(x,n);
  double m = mean(x,n);
  double s = stdev(x,n);
  cout << "mean = " << m << ", std dev = " << s << endl;
  for (int i=0; i<n; i++)
    cout << "x[" << i << "] = " << x[i]
      << ", z[" << i << "] = " << (x[i] - m)/s << endl;
}

6.34 int main()
{ double x[] = { 2.5, 4.5, 6.3, 6.7, 7.2, 7.5, 7.8, 9.9 }
  int n=8;
  print(x,n);
  double m = mean(x,n);
  double s = stdev(x,n);
  cout << "mean = " << m << ", std dev = " << s << endl;
  for (int i=0; i<n; i++)
  { double z = (x[i] - m)/s;
    cout << "x[" << i << "] = " << x[i]
      << ", z[" << i << "] = " << z;
    if (z >= 1.5) cout << " = A" << endl;
    else if (z >= 0.5) cout << " = B" << endl;
    else if (z >= -0.5) cout << " = C" << endl;
    else if (z >= -1.5) cout << " = D" << endl;
    else cout << " = F" << endl;
  }
}

```

```
6.35 void build_pascal(int p[][SIZE], int n)
{ assert(n > 0 && n < SIZE);
  for (int i=0; i<SIZE; i++)
    for (int j=0; j<SIZE; j++)
      if (i>n || j>i) p[i][j] = 0;
      else if (j==0 || j==i) p[i][j] = 1;
      else p[i][j] = p[i-1][j-1] + p[i-1][j];
}

6.36 double max_of_col(Matrix m, int n, int j)
{ double max=m[0][j];
  for (int i=1; i<n; i++)
    if (m[i][j]>max) max = m[i][j];
  return max;
}

double minimax(Matrix m, int n)
{ assert(n>0 && n < SIZE);
  double minimax=max_of_col(m,n,0);
  for (int j=1; j<n; j++)
    { double mm = max_of_col(m,n,j);
      if (mm<minimax) minimax = mm;
    }
  return minimax;
}
```

# 第 7 章 指针和引用

## 7.1 引用运算符

可以将计算机内存想象成一个非常大的字节数组。例如，一个计算机有 256MB 的 RAM（256 兆字节的随机存取存储器），实际上包含 268 435 456 ( $2^{28}$ ) 字节。作为一个数组，这些字节的下标范围从 0 到 268 435 455，每个字节的下标是它的内存地址，所以一个 256MB 的计算机内存单元的地址范围是从 0 到 268 435 455，十六进制数为 0x00000000 到 0xffffffff（参见附录 G）。图 7.1 表示了这个字节数组，并标明了它的十六进制地址。

变量声明将该变量与它的三个基本属性联系起来，这三个属性为：变量名、变量类型和变量的内存地址。例如，声明：

```
int n;
```

声明了变量名为 `n`，类型为 `int`，及 `n` 的值在内存中存放的地址。假设该地址为 0x0064fdf0，则可以将 `n` 看做图 7.2 所示：

变量本身用盒子来表示，变量名 `n` 在盒子的左边，变量的地址 0x0064fdf0 在盒子的上面，而变量类型在盒子下面。

在许多计算机中，变量类型 `int` 在内存中占用 4 个字节，所以上面的变量 `n` 占用 4 个字节的内存区域，如图 7.3 中的矩形阴影所示，使用了字节 0x0064fdf0，0x0064fdf1，0x0064fdf2，0x0064fdf3。注意目标的地址是它所存放的内存区域的第一个单元的地址。

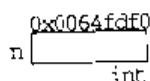


图 7.2 变量 `n` 的地址

如果变量被初始化，如下所示：

```
int n = 44;
```

则可以表示为图 7.4：

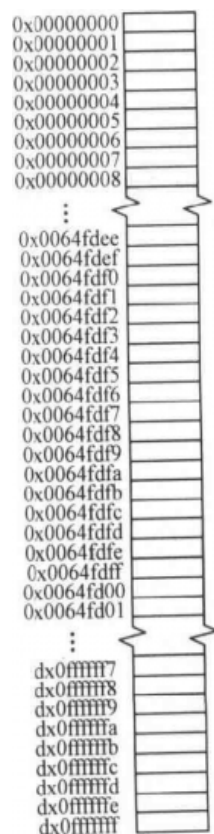


图 7.1 内存地址示例

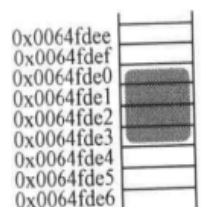


图 7.3 变量 `n` 占用 4 字节

变量的值被存放在分配给它的 4 个字节中如图 7.5 所示。

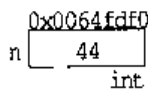


图 7.4 变量 n 示例



图 7.5 变量 n 示例

在 C++ 中, 可以使用引用运算符 & (也称为地址运算符) 来获得一个变量的地址, 表达式 &n 用来计算变量 n 的地址。

### 例 7.1 打印指针的值

```
int main ()
{ int n = 44;
  cout << "n = " << n << endl;    // 打印 n 的值
  cout << "&n = " << &n << endl; // 打印 n 的地址
}

n = 44
&n = 0x64fd0
```

输出说明 n 的地址为 0x64fd0。可以知道输出 0x64fd0 一定是一个地址, 因为它是用十六进制形式给出的, 标志是它以 0x 作为前缀。该地址的十进制数为 6 618 608 (参见附录 G)。

使用这种方式显示变量的地址并不是非常有用的, 引用运算符 & 还有更重要的用途。在第 5 章中说明了其中的一个用途: 在函数声明中指定引用参数, 该用途依赖于另一个用途——声明应用变量。

## 7.2 引用

引用是另一个变量的别名或同义字。声明引用的语法为:

类型 & 引用名 = 变量名;

其中, 类型是变量的类型, 引用名是引用的名字, 变量名是变量的名字。例如, 在下列声明中:

```
int& m = n; // r 是 n 的同义字
```

m 被声明为变量 n 的引用, 而 n 必须是已经声明的。

### 例 7.2 使用引用

```
int main ()
{ int n = 44;
  int& m = n; // r 是 n 的同义字
  cout << "n = " << n << ", m = " << m << endl;
```

```

--n;
cout << "n = " << n << ", m = " << m << endl;
m *= 2;
cout << "n = " << n << ", m = " << m << endl;
}
n=44, m=44
n=43, m=43
n=86, m=86

```

两个标志符 `n` 和 `m` 是同一个变量的不同名字，它们始终具有相同的值。减少 `n` 将会使 `n` 和 `m` 都变为 32，将 `n` 加倍将会使 `n` 和 `m` 都变为 64。

像常量一样，引用在声明时必须进行初始化，但与常量不一样的是，引用必须用变量来进行初始化，而不是一个数：

```
int& r n = 44; //错误：44 不是一个变量！
```

(某些编译器可能允许这样初始化，而给出一个警告信息：必须创建一个临时变量，并给它分配内存单元，以供被声明的引用来引用。)

虽然引用必须用变量来初始化，但引用并不是变量。变量是一个对象；即内存中用来存放可存取信息的一个连续字节区域，不同的变量必须占用不相邻的内存区域。

### 例 7.3 引用不是惟一的变量

```

int main ()
{ int n=44;
  int& m=n;    // r 是 n 的同义字
  cout << " &n = " << &n << ", &m = " << &m << endl;
  int& m2=n;    // r 是 n 的另一个同义字
  int& m3=m;    // r 是 n 的另一个同义字
  cout << "&m2 = " << &m2 << ", &m3 = " << &m3 << endl;
}
&n = 0x0064fde4, &m = 0x0064fde4
&m2 = 0x0064fde4, &m3 = 0x0064fde4

```

第一个输出语句说明 `n` 和 `m` 具有相同的地址：0x0064fde4，因此，它们仅仅是同一个对象的不同名字。第二个输出语句说明一个对象可以有几个引用，并且一个引用的引用与一个对象的引用是相同的。在上例中，只有一个对象：一个名为 `n`、地址为 0x0064fde4 的 `int` 型对象，名字 `m`、`m2` 和 `m3` 都是这个对象的引用，如图 7.6 所示。

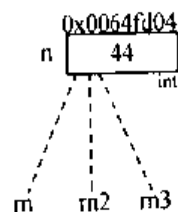


图 7.6 指向同一对象的不同指针

在 C++ 中，引用运算符 `&` 有两个不同的用途。当用做一个对象名的前缀时，它构成一个表达式来计算该对象的地址。当用做一个类型 `T` 的后缀时，它给一个派生类型“`T` 的引用”命名。例如，`int&` 是类型“`int` 的引用”。所以在例 7.3 中，`n` 被声明为类型是 `int`，并且 `m` 被声明为类型 `int` 的引用。

C++ 实际上有 5 种派生类型。如果 `T` 是一种类型，则 `const T` 是派生类型“常量 `T`”，`T()` 是派生类型“函数 `T`”，`T[]` 是派生类型“数组 `T`”，`T&` 是派生类型“`T` 的引用”，而

$T^*$  是派生类型“指向  $T$  的指针”。

在更多时候，引用是用于引用参数（参见第 5.2 节）。引用参数的工作方式与引用变量相同：它们仅仅是其他变量的同义字。实际上，函数的引用参数就像一个作用域限制在函数内部的引用变量中。

## 7.3 指针

引用运算符  $\&$  返回使用它的变量的内存地址，由此在例 7.1 中打印出内存地址。也可以将地址存放在另一个变量中，存放地址的变量类型称为指针。指针变量的类型为“指向  $T$  的指针”，其中  $T$  为指针所指向的对象的类型。如第 7.2 节所述，该派生类型被表示为  $T^*$ 。例如，一个  $\text{int}$  类型变量的地址可以被存储在一个  $\text{int}^*$  类型的指针变量中。

### 例 7.4 使用指针变量

本例定义了  $\text{int}$  型的变量  $n$  和  $\text{int}^*$  型的变量  $pn$ ，如图 7.7 所示：

```
int main ()
{ int n=44;
  cout << "n = " << n << ", &n = " << &n << endl;
  int * pn=&n; // pn 保存 n 的地址
  cout << "          pn = " << pn << endl;
  cout << "&pn = " << &pn << endl;
```

```
n = 44, &n = 0x0064fddc
pn = 0x0064fddc
&pn = 0x0064fde0
```

变量  $n$  初始化为 44，其地址为  $0x0064fddc$ ，变量  $pn$  初始化为  $\&n$ ，即  $n$  的地址，所以  $pn$  的值为  $0x0064fddc$ ，正如第二个输出语句所示。但是  $pn$  是一个单独的变量，如第三个输出语句所示：它有一个不同的地址  $0x0064fde0$ ，如图 7.8 所示。

变量  $pn$  被称为一个“指针”，是因为其值“指向”另一个变量的存储位置，一个指针的值是一个地址，该地址依赖于运行程序的那台计算机的状态。很多时候，地址的实际值（这里为  $0x0064fddc$ ）不是程序员所关心的问题，所以示意图通常简单地画成如图 7.9 所示的样子，它抓住了  $n$  和  $pn$  的本质特征： $pn$  是一个指向  $n$  的指针，而  $n$  的值为 44。一个指针可以被想象成一个“位置指示器”：它指明另一个对象的位置。

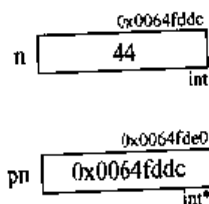


图 7.7 变量  $n$  与指针  $pn$

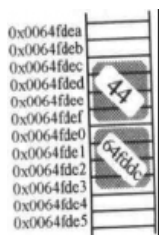


图 7.8 变量  $n$  与指针  $pn$  在内存中

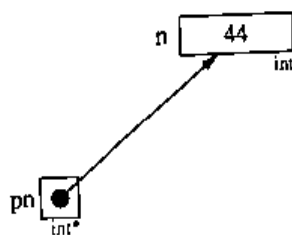


图 7.9  $n$  与  $pn$  的关系

## 7.4 取值操作符

如果 `pn` 指向 `n`，可以从 `p` 直接得到 `n` 的值；表达式 `*pn` 评估 `n` 的值。这种评估叫做“取值指针”`pn`，符号 `*` 叫做取值运算符。

### 例 7.5 取指针的值

本例比例 7.4 多一行代码，但运行效果相同：

```
int main ()
{ int n=44;
  cout << "n = " << n << ", &n = " << &n << endl;
  int * pn = &n; // pn 保存 n 的地址
  cout << "      pn = " << pn << endl;
  cout << "&pn = " << &pn << endl;
  cout << "*pn = " << *pn << endl;
}
```

```
n = 44, &n = 0x0064fddc
      pn = 0x0064fddc
&pn = 0x0064fde0
*pn = 44
```

该例说明 `*pn` 是 `n` 的别名：它们具有相同的值 44。

### 例 7.6 指针的指针

本例是在例 7.4 的基础上编写的：

```
int main ()
{ int n=44;
  cout << "      n = " << n << endl;
  cout << "      &n = " << &n << endl;
  int * pn = &n; // pn 保存 n 的地址
  cout << "      pn = " << pn << endl;
  cout << "      &pn = " << &pn << endl;
  cout << "      *pn = " << *pn << endl;
  int ** ppn = &pn; // ppn 保存 pn 的地址
  cout << "      ppn = " << ppn << endl;
  cout << "      &ppn = " << &ppn << endl;
  cout << "      *ppn = " << *ppn << endl;
  cout << "      **ppn = " << **ppn << endl;
}
```

```
n = 44
&n = 0x0064fd78
pn = 0x0064fd78
&pn = 0x0064fd7c
*pn = 44
ppn = 0x0064fd7c
&ppn = 0x0064fd80
*ppn = 0x0064fd78
**ppn = 44
```

变量 `ppn` 指向 `pn`, 而 `pn` 指向 `n`, 所以 `*ppn` 是 `pn` 的别名, 就如 `*pn` 是 `n` 的别名, 因此 `**ppn` 也是 `n` 的别名。

注意在例 7.6 中, 三个变量 `n`、`pn` 和 `ppn` 具有不同的类型: `int`, `int*`, `int**`。一般而言, 如果 `T1` 和 `T2` 是不同的类型, 则它们的派生类型也具有不同的类型, 所以即使 `pn` 和 `ppn` 都是指针, 它们并不是同一个类型。`pn` 的类型为指向 `int` 的指针, 而 `ppn` 的类型为指向 `int*` 的指针, 如图 7.10 所示。

图 7.10 二级指针示例

引用运算符 `&` 和取值运算符 `*` 是相反的: `n == *p`, 而 `p == &n`, 也可以表示为: `&*n == n` 和 `&*p == p`。

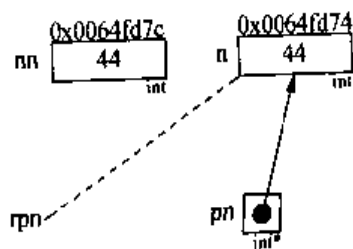
### 例 7.7 引用与取值相反

本例也是在例 7.4 基础上编写的。

```
int main ()
{ int n = 44;
  cout << "    n = " << n << endl;
  cout << "    &n = " << &n << endl;
  int * pn = &n;    // pn 保存 n 的地址
  cout << "    pn = " << pn << endl;
  cout << "    &pn = " << &pn << endl;
  cout << "    *pn = " << *pn << endl;
  int nn = *pn;    // nn 是 n 的复制品
  cout << "    nn = " << nn << endl;
  cout << "    &nn = " << &nn << endl;
  int& rpn = *pn;    // rpn 是 n 的引用
  cout << "    rpn = " << rpn << endl;
  cout << "    &rpn = " << &rpn << endl;
```

```
n = 44
&n = 0x0064fd74
pn = 0x0064fd74
&pn = 0x0064fd78
*pn = 44
nn = 44
&nn = 0x0064fd7c
rpn = 44
&rpn = 0x0064fd74
```

这里 `p` 指向整型数 `pn`, 而 `rpn` 是一个引用, `rpn` 被初始化为 `pn` 所指向的值, 所以 `pn` 引用 `n` 而 `rpn` 取 `pn` 的值, 因此 `rpn` 是 `n` 的别名, 即它们是同一个对象的不同名字。输出证明了这一点: `n` 和 `rpn` 具有相同的地址 `0x0064fd74`, 如图 7.11 所示。

图 7.11 `n`, `rpn`, `pn` 的关系



## 7.5 派生类型

如同引用运算符 &，取值运算符 \* 有两个不同的用途。当用做一个指向对象的指针的前缀时，它构成一个表达式来计算对象的值。当用做一个类型 T 的后缀时，它命名派生类型“指向 T 的指针”。例如，int \* 是类型“指向 int 的指针”。

如上所述，在 C++ 中有 5 种派生类型。下面列出一些派生类型的声明：

```
const int C = 33;           // int 型常量
int& m = n;                 // int 型引用
int * pn = &n;              // 指向 int 型的指针
int a [] = {33, 66};        // int 型数组
int f [] = {return 33;};    // 返回 int 型的函数
```

一个派生类型可以从任意其他类型派生来，所以可能有许多联合类型：

```
int * const Pn = 44;        // 指向 int 型的指针常量
const int * pN = &N         // 指向 int 型常量的指针
const int * const PN = &N;  // 指向 int 型常量的指针常量
float& ar [] = {x, y};      // 两个 float 型引用构成的数组
float * ap [] = {&x, &y};   // 两个指向 float 型的指针构成的指针数组
long& r () {return n;};      // 返回 long 型引用的函数
long * p () {return &n;};    // 返回指向 long 型的指针的函数
long (* pf) () {return 44;}; // 指向返回 long 型数的函数的指针
```

一些派生类型需要 typedef 的辅助：

```
typedef char Word [255];     // 有 255 个 char 型元素的数组
Word& pa = a;               // 有 255 个 char 型元素的数组的引用
Word * pa = &a              // 指向有 255 个 char 型元素数组的指针
```

## 7.6 对象和左值

Ellis 所著的《注释 C++ 引用手册》中说到：“一个对象是一个存储区，一个左值是引用一个对象或函数的表达式。”最初，术语“左值”和“右值”是出现在赋值语句的左边和右边的表达式的引用，但是现在“左值”的概念更为一般化了。

左值的最简单的例子是对象（即变量）名：

```
int n;
n = 44; // n 是一个左值
```

是数值而不是左值的最简单的例子是：

```
44 = n; // 错误：44 不是一个左值
```

但是符号常量是左值：

```
const int MAX = 65535;    // MAX 是一个左值
```

虽然符号常量不能出现在一个赋值语句的左边：

```
MAX = 21024;    // 错误：MAX 是常量
```

可以出现在赋值语句左边的左值称为可变左值；那些不能出现在赋值语句左边的左值称为不变左值。常量是不变左值；变量是不变左值。其余可变左值的例子包括下标变量和取值指针：

```
int a [8];
a [5] = 22;    // a [5] 是一个可变左值
int * p = &n;
*p = 77;       // *p 是一个可变左值
```

其他不变左值的例子包括数组、函数和引用。

一般而言，左值是其地址可访问的值。由于声明一个引用变量时需要知道地址，C++ 的语法要求这样声明一个左值：

类型 & 引用名 = 左值；

例如，下面是一个合法的引用声明：

```
int& r = n;    // 正确：n 是一个左值
```

但是下面的声明是非法的：

```
int& r = 44;           // 错误：44 不是一个左值
int& r = n++;          // 错误：n++ 不是一个左值
int& r = cube (n);     // 错误：cube (n) 不是一个左值
```

## 7.7 返回引用

函数的返回类型可以是一个引用，它的返回值是一个不属于函数内部的左值，其含义为：返回值实际上是一个在函数终止后依然存在的左值的引用。因此，该返回值可以像其他的左值一样使用；例如，可用在赋值语句的左边。

### 例 7.8 返回引用

```
int& max (int& m, int& n);    // 返回类型是指向 int 的引用
{ return (m > n ? m : n);    // m 和 n 是非局部引用
}
int main ()
{ int m = 44, n = 22;
  cout << m << ", " << n << ", " << max (m, n) << endl;
  max (m, n) = 55;           // 将 m 的值从 44 变为 55
  cout << m << ", " << n << ", " << max (m, n) << endl;
}
```

```
44, 22, 44
55, 22, 55
```

`max()` 函数返回传递给它的两个变量的最大值的引用。由于返回值是一个引用，表达式 `max(m, n)` 的作用就像一个 `m` 的引用（因为 `m` 大于 `n`），所以将 55 赋值给 `max(m, n)` 就相当于将 55 赋值给 `m` 本身。

### 例 7.9 使用函数作为数组下标

```
float& component (float * v, int k)
| return v[k-1];
|
int main ()
| float v[4];
  for (int k = 1; k <= 4; k++)
    component (v, k) = 1.0/k;
  for (int i = 0; i < 4; i++)
    cout << "v[" << i << "] = " << v[i] << endl;
```

```
v[0] = 1
v[1] = 0.5
v[2] = 0.333333
v[3] = 0.25
```

函数 `component()` 允许使用科学的“1 阶下标”来访问所有的矢量，而不是使用默认的“0 阶下标”来访问，所以赋值语句 `component(v, k) = 1.0/k` 实际上是赋值语句 `v[k+1] = 1.0/k`。在第 10 章将看到用更好的方法来完成相同的操作。

## 7.8 数组和指针

虽然指针类型不是整数型，一些整数型的算术运算符可以被用于指针，这种算术运算的结果是使该指针指向另一个内存单元。地址的真正改变依赖于指针所指向的基本类型的大小。

指针可以像整数一样增加和减少，然而，指针值的增量和减量与该指针所指向的对象的大小相等。

### 例 7.10 使用指针来遍历一个数组

本例说明如何使用一个指针来遍历一个数组：

```
int main ()
| const int SIZE = 3;
  short a[SIZE] = {22, 33, 44};
  cout << "a = " << a << endl;
  cout << "sizeof (short) = " << sizeof (short) << endl;
  short * end = a + SIZE;      // 将 SIZE 转成偏移量 6
```

```

short sum = 0;
for (short * p = a; p < end; p++)
{
    sum += *p;
    cout << "\t p = " << p;
    cout << "\t *p = " << *p;
    cout << "\t sum = " << sum << endl;
}
cout << "end = " << end << endl;
}

```

```

a=0x3fffd1a
sizeof (short) =2
    p=0x3fffd1a    *p=22    sum=22
    p=0x3fffd1c    *p=33    sum=55
    p=0x3fffd1e    *p=44    sum=99
end=0x3fffd20

```

第二个输出语句说明了在这台机器中 short 整数型的整数占用了 2 个字节。由于 p 是一个指向 short 型的指针，每次它加 1，将前进 2 个字节指向数组中的下一个 short 整数型的整数。由此，sum += \*p 计算这些整数的和。如果 p 是一个指向 float 型的指针，而 sizeof (double) 为 8 字节，则每次 p 加 1，将前进 8 个字节。

例 7.10 说明当一个指针递增时，其增值是由它所指向的对象的字节数。例如：

```

float a [8];
float * p=a;    // p 指向 a [0]
++p;            // 根据 sizeof (float) 增加 p 的值

```

如果 float 型占用 4 字节，则 ++p; 将 p 值加 4，并且 p += 5，将 p 值加 20。这就说明为什么可以遍历数组：首先将一个指针初始化为指向数组的第一个元素，然后反复地递增该指针，每次递增都使指针指向数组的下一个元素。

也可以使用一个指针在数组中实现随机访问。例如，首先将指针初始化为 a [0]，然后将指针加 5，就可以访问 a [5]：

```

float * p=a;    // p 指向 a [0]
p+=5;           // 现在 p 指向 a [5]

```

所以一旦指针被初始化为数组的起始地址，它就可以像一个下标一样使用。

警告：在 C++ 中，可以访问甚至修改未被分配的内存单元，但这样做是危险的，一般来讲应该尽量避免。例如：

```

float a [8];
float * p=a [7];    // p 指向数组的最后一个元素
++p;                // 现在 p 指向超出最后一个元素的内存单元
*p=22.2;            // 故障！

```

下例说明数组和指针的密切联系：数组名本身是一个指向第一个元素的 const 型指针，同时该例说明指针可以进行比较。

**例 7.11 检查数组元素的地址**

```
int main ()
{ short a [] = {22, 33, 44, 55, 66};
  cout << "a = " << a << ", *a = " << *a << endl;
  for (short * p = a; p < a + 5; p++)
    cout << "p = " << p << ", *p = " << *p << endl;
}
```

```
a=0x3fffd08,    *a=22
p=0x3fffd08,    *p=22
a=0x3fffd0a,    *p=33
a=0x3fffd0c,    *a=44
a=0x3fffd0e,    *a=55
a=0x3fffd10,    *a=66
```

开始,  $a$  与  $p$  是相同的, 它们都是指向 `short` 型的指针且具有相同的值 (`0x3fffd08`)。由于  $a$  是一个指针常量, 它不能递增来遍历数组。相反, 可以递增  $p$  并且使用退出条件  $p < a + 5$  来终止循环。 $a + 5$  计算的是十六进制的地址  $0x3fffd08 + 5 * \text{sizeof}(\text{short}) = 0x3fffd08 + 5 * 2 = 0x3fffd08 + 0xa = 0x3fffd12$ , 所以, 在  $p < 0x3fffd12$  时执行循环。

数组下标运算符 `[]` 与取值运算符 `*` 等价, 它们以相同的方式提供数组的随机访问:

```
a[0] == *a;
a[1] == *(a+1);
a[2] == *(a+2);
```

依此类推, 所以数组可以被这样遍历:

```
for (int i = 0; i < 8; i++)
  cout << * (a+i) << endl;
```

下面的例子说明, 怎样将指针和整数联合使用来在内存中前后移动。

**例 7.12 模式匹配**

在本例中, `loc()` 函数在数组  $a1$  中的前  $n1$  个元素中查找在数组  $a2$  中的前  $n2$  个元素中存放的整数串, 如果找到, 则返回一个指针, 指向在  $a1$  中  $a2$  开始的单元; 否则, 返回空指针。

```
short * loc (short * a1, short * a2, int n1, int n2)
{ short * endl = a1 + n1;
  for (short * p1 = a1; p1 < endl; p1++)
    if (*p1 == *a2)
      { int j;
        for (j = 0; j < n2; j++)
          if (p1[j] != a2[j]) break;
        if (j == n2) return p1;
      }
  return 0;
}
```

```

int main ()
{ short a1 [9] = {11, 11, 11, 11, 11, 22, 33, 44, 55};
  short a2 [5] = {11, 11, 11, 22, 33};
  cout << "Array a1 begins at location\t" << a1 << endl;
  cout << "Array a2 begins at location\t" << a2 << endl;
  short * p = loc (a1, a2, 9, 5);
  if (p)
  { cout << "Array a2 found at location\t" << p << endl;
    for (int i = 0; i < 5; i++)
      cout << "\t" << &p [i] << ": " << p [i]
        << "\t" << &a2 [i] << ": " << a2 [i] << endl;

    else cout << "Not found. \n";
  }
}

```

```

Array a1 begins at location      0x3fffd12
Array a2 begins at location      0x3fffd08
Array a2 begins at location      0x3fffd16
0x3fffd16: 11    0x3fffd08: 11
0x3fffd18: 11    0x3fffd0a: 11
0x3fffd1a: 11    0x3fffd0c: 11
0x3fffd1c: 22    0x3fffd0e: 22
0x3fffd1e: 33    0x3fffd10: 33

```

模式匹配算法使用双层循环。外层循环使用指针  $p1$  控制,  $p1$  指向数组  $a1$  中的元素, 而内层循环将在数组  $a2$  中寻找一个匹配。内层循环使用整数  $j$  控制,  $j$  用于比较两个数组中相对应的元素。只要找到一个不匹配, 则内层循环终止, 并且外层循环将  $p1$  递增, 从  $a1$  的下一个元素开始继续寻找匹配。如果内层循环正常结束, 则条件 ( $j = n2$ ) 为真, 并且  $p1$  返回正确的位置值。

测试程序证明通过检查实际的地址真的可以找到匹配。

### 例 7.13 new 运算符

当这样声明一个指针时:

```
float * p;    // p 是一个指向 float 型的指针
```

它仅仅是将内存分配给指针本身, 指针的值将是某个内存地址, 但是该地址的内存单元还没有被分配, 这意味着这个单元可能已经被其他的变量使用。此时,  $p$  并没有初始化, 它没有指向任何已分配的内存单元, 试图访问它所指向的单元将会出错。例如:

```
p = 3.14159;    // 错误: 没有内存单元被分配给 *p
```

避免这个问题的方法是在声明指针的同时进行初始化。例如:

```

float x = 3.14159;    // x 包含值 3.14159
float * p = &x;       // p 包含 x 的地址
cout << p;            // 正确: *p 已经被分配

```

此时, 访问 `*p` 就没什么问题了, 因为当声明 `x` 时, 自动分配存放 `float` 型数 `3.14159` 的内存单元 `p` 指向同一个单元。

另一种避免上面问题的方法是使用虚悬指针将内存分配给该指针本身, 这时要使用 `new` 运算符。

```
float * q;  
q = new float;           // 分配 1 个 float 的内存单元  
*q = 3.14159;           // 正确: *q 已经被分配了内存空间
```

`new` 运算符返回未被分配的 `s` 字节内存区域的地址, 其中 `s` 是一个 `float` 型数的大小 (例如, `sizeof (float)` 为 4 字节)。将该地址分配给 `q` 可以保证 `*q` 当前没有被任何其他变量使用。

前两个语句可以合并, 因此, 可以在声明 `q` 的同时进行初始化:

```
float * q = new float;
```

注意, 使用 `new` 运算符来初始化 `q` 仅仅是初始化指针本身, 而不是它所指向的内存单元。可以在同一个语句中完成这两件事:

```
float * q = new float (3.14159);  
cout << q;           // 正确: q 和 *q 都被初始化
```

当没有足够的内存空间可分配时, 则 `new` 运算符将返回 0 (空指针)。例如:

```
double * p = new double;  
if (p == 0) abort ();    // 分配错误: 没有足够的内存空间  
else *p = 3.141592658979324;
```

这里调用 `abort ()` 函数来防止取空指针的值。

再考虑分配内存的两种方法:

```
float x = 3.14159;        // 分配命名的内存  
float * p = new float (3.14159); // 分配未命名的内存
```

在第一种情况下, 在编译时内存被分配给命名的变量 `x`; 在第二种情况下, 在运行时内存被分配给一个通过 `*p` 访问的未命名的对象。

### 例 7.14 delete 运算符

`delete` 运算符与 `new` 运算符的操作相反, 用来释放被分配的内存空间, 它仅能用于已经被 `new` 运算符明确分配了内存的指针。例如:

```
float * q = new float (3.14159);  
delete q;           // 释放 q  
*q = 2.71828;       // 错误: q 已经被释放
```

释放 `q` 指的是释放 `sizeof (float)` 字节的内存区域, 使它可以分配给其他对象。一旦 `q` 被释放, 在它再次被分配之前就不能再使用了。一个被释放的指针, 也称为虚悬指针, 如同

一个未被初始化的指针，它不指向任何对象。

一个指向常量的指针不能被释放。例如：

```
const int *p = new int;
delete p;           // 错误：不能释放指向常量的指针
```

这个限制是与不能修改常量的基本原则相一致的。

一般不提倡对基本类型（char, int, float, double 等）使用 delete 运算符，因为这样做有可能造成严重错误。例如：

```
float x = 3.14159;    // x 包含值 3.14159
float * p = &x;       // p 包含 x 的地址
delete p;             // 危险：p 没有用 new 分配内存空间
```

这样做将释放变量 x 的内存空间，是一个非常难以调试的错误。

## 7.9 动态数组

在编译时，一个数组名实际上就相当于一个指针常量。例如：

```
float a [20];        // a 是一个指向 20 个 float 的内存单元的指针常量
float * const p = new float [20];    // p 与 a 相同
```

这里，a 和 p 都是一个指向 20 个 float 的内存单元的指针常量。a 的声明称为静态绑定，因为它是在编译时分配内存空间的；在程序运行时即使该数组没被使用，符号 a 也是与分配给它的内存空间绑定在一起的。

相反，可以使用一个不是常量的指针来将内存分配推迟到程序运行时，这通常称为运行时绑定或动态绑定。例如：

```
float * p = new float [20];
```

这样声明的数组称为动态数组。

比较定义数组的两种方法：

```
float a [20];           // 静态数组
float * p = new float [20]; // 动态数组
```

静态数组 a 是在编译时创建的。分配给它的内存空间会保持到程序运行结束。动态数组 p 是在运行时创建的；只有当该声明语句运行时才给它分配内存空间，更进一步说，一旦对它使用 delete 运算符，分配给数组 p 的内存空间就立即被释放。例如：

```
delete [] p;           // 释放数组 p
```

注意，这时必须使用下标运算符 []，因为 p 是一个数组。

### 例 7.15 使用动态数组

下例中的 get () 函数创建一个动态数组：



```

void get (double* &a, int&n)
{ cout << "Enter number of items: ";   cin >> n;
  a = new double [n];
  cout << "Enter " << n << " items, one per line: \n";
  for (int i = 0; i < n; i++)
    { cout << "\t" << i+1 << ": ";
      cin >> a [i];
    }
}

void print (double* a, int n)
{ for (int i = 0; i < n; i++)
  cout << a [i] << " ";
  cout << endl;
}

int main ()
{ double* a;      // a 只是一个未分配的指针
  int n;
  get (a, n);     // 现在 a 是 n 个双精度型数组
  print (a, n);
  delete [] a;    // 现在 a 又是一个未分配的指针
  get (a, n);     // 现在 a 是 n 个双精度型数组
  print (a, n);
}

```

```

Enter number of items: 4
Enter 4 items, one per line:
    1: 44.4
    2: 77.7
    3: 22.2
    4: 88.8
44.4 77.7 22.2 88.8
Enter number of items: 2
Enter 2 items, one per line:
    1: 3.33
    2: 9.99
3.33 9.99

```

在 `get ()` 函数中, 输入 `n` 的值后, `new` 运算符分配 `n` 个 `double` 型的内存空间, 所以数组是在程序运行时“在飞行中”被创建的。

在使用 `get ()` 创建另一个数组 `a` 之前, 使用 `delete` 运算符释放了当前数组。注意, 在释放一个数组时必须指定下标运算符 `[]`。

注意, 数组参数 `a` 是一个通过引用传递的指针:

```
void get (double*&a, int&n)
```

这是必要的, 因为 `new` 运算符将修改 `a` 的值, 而 `a` 是最近被分配内存空间的数组的第一个元素的地址。

## 7.10 通过指针使用 `const`

一个指向常量的指针与一个指针常量不同, 下例说明了这种区别。

### 例 7.16 指针常量和指向常量的指针

本例是一个程序片断，声明了 4 个变量：一个指针 `p`，一个指针常量 `cp`，一个指向常量的指针 `pc` 和一个指向常量的指针常量 `cpc`：

```
int n = 44;           // 一个 int 型数
int * p = &n;         // 一个指向 int 型数的指针
++ (*p);              // 正确：将 *p 加 1
++ p;                 // 正确：将指针 p 加 1
int * const cp = &n;  // 一个指向 int 型数的指针常量
++ (*cp);             // 正确：将 *cp 加 1
++ cp;                // 非法：指针 cp 是常量
const int k = 88;     // 一个 int 型常量
const int * pc = &k;  // 一个指向 int 型常量的指针
++ (*pc);             // 非法：*pc 是一个常量
++ pc;                // 正确：将指针 pc 加 1
const int * const cpc = &k; // 一个指向 int 型常量的指针常量
++ (*cpc);            // 非法：*cpc 是一个常量
++ cpc;               // 非法：指针 cpc 是一个常量
```

注意，引用运算符 `*` 用于声明语句时，它的两边可以有一个空格，也可以没有空格。因此，下面的三个声明语句是等价的：

```
int * p;      // 指明 p 的类型为 int *（指向 int 型的指针）
int * p;      // 有时为清楚起见所采用的风格
int * p;      // 旧的 C 语言风格
```

## 7.11 指针数组和指向数组的指针

一个数组的元素可以为指针，下面是一个类型为 `double` 的 4 个指针组成的数组：

```
double * p[4];
```

它的元素可以像其他的指针一样分配内存：

```
p[0] = new double (2.718281828459045);
p[1] = new double (3.141592653589793);
```

可以按图 7.12 所示看待这些数组。

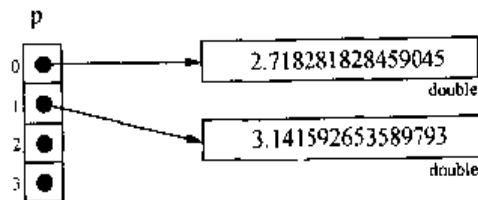


图 7.12 指针数组

下例是指针数组的一个非常有用的应用，它说明了怎样通过指向元素的指针来间接对一个数据序列排序，而不是直接移动元素本身。该例与习题 5.12 中的间接冒泡排序等价。

### 例 7.17 间接冒泡排序

```
void sort (float * p[], int n)
{ float * temp;
```

```

for (int i=1; i<n; i++)
    for (int j=0; j<n-i; j++)
        if (*p[j] > *p[j+1])
            | temp=p[j];
            p[j] =p[j+1];
            p[j+1] =temp;
    }
}

```

在内层的每次循环中，如果相邻指针所指向的 float 数不满足升序，则交换指针。

## 7.12 指向指针的指针

一个指针可以指向另一个指针，例如：

```

char c = 't';
char * pc = &c;
char * * ppc = &pc;
char * * * pppc = &ppc;
* * * pppc = 'w'; // 将 c 的值改为 'w'

```

可以用图 7.13 中所示的方法看待这些变量。

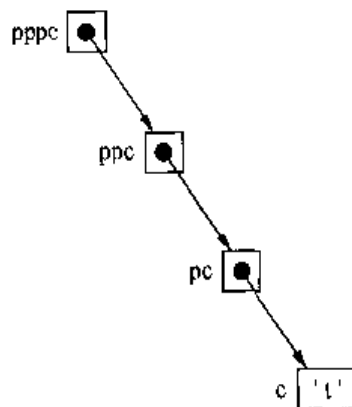


图 7.13 指向指针的指针

## 7.13 指向函数的指针

像一个数组名一样，函数名实际上是一个指针常量，可以将它的值看做实现该函数的代码的地址。

一个指向函数的指针的值就是函数名地址。由于函数名本身是一个指针，所以一个指向函数的指针是一个指向指针常量的指针。例如：

```

int f (int);           // 声明函数 f
int (* pf) (int);      // 声明函数指针 pf
pf = &f;               // 将 f 的地址赋给 pf

```

可以用图 7.14 中所示的方法看待这些变量。

函数指针的值是允许定义函数的函数的值，这是通过将一个函数指针作为一个参数传递给另一个函数实现的。

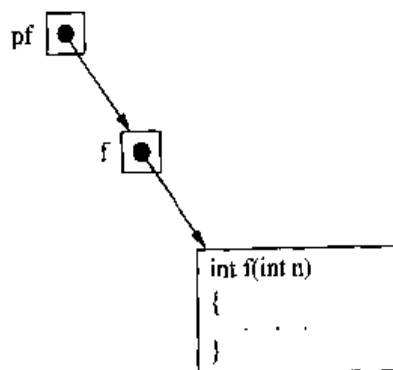


图 7.14 指向函数的指针

### 例 7.18 函数的和

sum () 函数有两个参数：函数指针 pf 和整数 n。

```

int sum (int (*) (int), int);
int square (int);
int cube (int);

```

```
int main ()
{ cout << sum (square, 4) << endl;    // 1 + 4 + 9 + 16
  cout << sum (cube, 4) << endl;      // 1 + 8 + 27 + 64
}
```

调用 `sum (square, 4)` 计算并返回 `square (1) + square (2) + square (3) + square (4)` 的和。因为 `square (k)` 计算并返回  $k * k$ ，所以 `sum ()` 函数返回  $1 + 4 + 9 + 16 = 30$ 。

下面是函数的定义和输出结果：

```
int sum (int (*pf) (int k), int n)
{ // 返回 f (0) + ..... 的和
  int s = 0;
  for (int i = 1; i <= n; i++)
    s += (*pf) (i);
  return s;
}

int square (int k)
{ return k * k;
}
```

```
int cube (int k)
{ return k * k * k;
}
```

```
30
100
```

`sum ()` 函数从 1 到 n 计算 pf 所指向的函数，并返回它们的和。

注意，在 `sum ()` 函数的参数列表中声明函数指针参数 pf 时，需要哑变量 k。

## 7.14 NUL、NULL 和 void

常量 0 的类型为 int，然而这个符号 0 可以被赋值给所有的基本类型：

```
char c = 0;          // 将 c 初始化为 char 型的 '0'
short d = 0;         // 将 d 初始化为 short int 型的 0
int n = 0;           // 将 n 初始化为 int 型的 0
unsigned u = 0;      // 将 u 初始化为 unsigned int 型的 0
float x = 0;         // 将 x 初始化为 float 型的 0.0
double z = 0;        // 将 z 初始化为 double 型的 0.0
```

在每种情况下，对象被初始化为数 0。在类型为 char 时，字符 c 成为空字符，被赋值为 `'\0'` 或 `NUL`，它是 ASCII 码值为 0 的字符。

指针的类型是内存地址，这些地址必须在分配给该执行程序的内存空间的范围内，地址 `0x0` 是例外，它被称为空 (NULL) 指针。在下面的各种类型中，同样的常量被赋给指针：

```
char * pc = 0;       // 初始化 pc 为空指针
```

```
short * pd = 0;    // 初始化 pd 为空指针
int * pn = 0;      // 初始化 pn 为空指针
unsigned * pu = 0; // 初始化 pu 为空指针
float * px = 0;    // 初始化 px 为空指针
double * pz = 0;   // 初始化 pz 为空指针
```

不能取空指针的值，这是一个常见但很严重的错误：

```
int * p = 0;
*p = 22;    // 错误：不能取空指针的值
```

一个避免上面错误的合理方法是在对一个指针取值前先测试这个指针：

```
if (p) *p = 22; // 正确
```

这个语句测试条件 ( $p \neq \text{NULL}$ )，因为只有当  $p$  非零时，该条件才为真。

`void` 表示一个特殊的基本类型。与所有其他的基本类型不同，`void` 只能用于一个派生类型。例如：

```
void x;    // 错误：没有对象可以具有类型 void
void * p;  // 正确
```

类型 `void` 最常见的用途是说明没有返回值的函数。例如：

```
void swap (double&, double&);
```

`void` 的另一个不同的用途是声明一个指向未知类型的对象的指针：

```
void * p = q;
```

该用途更多地用于使用底层 C 程序来对硬件资源进行操作。

## 复 习 题

- 7.1 如何取得一个变量的内存地址？
- 7.2 如何访问地址存放在一个指针变量中的内存单元的内容？
- 7.3 说明下面两个声明的区别：

```
int n1 = n;
int& n2 = n;
```

- 7.4 说明下面两个引用运算符 `&` 的不同用法：

```
int& r = n;
p = &n;
```

- 7.5 说明下面两个取值运算符 `*` 的不同用法：

```
int * q = p;
n = *p;
```

7.6 下面的说法是否正确? 为什么?

- a. 如果  $(x == y)$  则  $(\&x == \&y)$
- b. 如果  $(x == y)$  则  $(*x == *y)$

7.7 a. 什么是“虚悬指针”?

- b. 对一个虚悬指针取值会带来什么样的严重后果?
- c. 如何避免这样的严重后果?

7.8 下面的代码有什么错误?

```
int& r = 2;
```

7.9 下面的代码有什么错误?

```
int * p = &44;
```

7.10 下面的代码有什么错误?

```
char c = 'w';  
char p = &c;
```

7.11 为什么例 7.6 中的变量 `ppn` 不能这样声明:

```
int * * ppn = &&n;
```

7.12 “静态绑定”和“动态绑定”的区别是什么?

7.13 下面的代码有什么错误?

```
char c = 'w';  
char * p = c;
```

7.14 下面的代码有什么错误?

```
short a [32];  
for (int i = 0; i < 32; i++)  
    *a++ = i*i
```

7.15 判断以下代码执行后每个指定的变量的值。假设每个整型变量占用 4 字节, 并且 `m` 在内存中的地址为 `0x3fffd00`。

```
0x3fffd00.  
int m = 44;  
int * p = &m;  
int& r = m;  
int n = (*p)++;  
int * q = p - 1;  
r = * (--p) + 1;  
++ * q;
```

a. `m`

b. `n`

```
c. &md. *p
```

```
e. r
```

```
f. *c
```

7.16 将下面的变量按照可变左值、不变左值和非左值分类:

```
a.double x = 1.23
```

```
b.4.56 * x + 7.89
```

```
c.const double Y = 1.23;
```

```
d.double a [8] = {0.0};
```

```
e.a [5]
```

```
f.double f () { return 1.23;}
```

```
g.f (1.23)
```

```
h.double& r = x;
```

```
i.double* p = &x;
```

```
j.* p
```

```
k.const double* p = &x;
```

```
l.double* const p = &x;
```

7.17 下面的代码有什么错误?

```
float x = 3.14159;
```

```
float* p = &x;
```

```
short d = 44;
```

```
short* q = &d;
```

```
p = q;
```

7.18 下面的代码有什么错误?

```
int* p = new int;
```

```
int* q = new int;
```

```
cout << "p = " << p << ", p+q = " << p + q << endl;
```

7.19 使用空指针惟一能做的事是什么?

7.20 在下面的声明中,说明 p 的类型,并描述如何使用 p:

```
double* * * * p;
```

7.21 如果 x 的地址为 0x3fffd1c, 则下面的代码中 p 和 q 的值是什么?

```
double x = 1.01;
```

```
double* p = &x;
```

```
double* q = p + 5;
```

7.22 如果 p 和 q 分别为指向 int 型的指针, n 为一个 int 型数, 则以下代码中哪个是非法的:

```
a. p + q
```

b.  $p - q$   
 c.  $p + n$   
 d.  $p \cdot n$   
 e.  $n + p$   
 f.  $n - q$

7.23 一个数组总是一个指针常量的说法的含义是什么?

7.24 当只把数组第一个元素的地址传递给函数时, 为什么函数可以访问这个数组的所有元素?

7.25 对于数组  $a$  和  $\text{int}$  型数  $x$ , 说明为什么下面的条件为真:

```
a[i] == * (a + i);
* (a + i) == i[a];
a[i] == i[a];
```

7.26 说明下面两个声明的区别:

```
double * f();
double (* f)();
```

7.27 写出下面的声明:

- 声明一个大小为 8 的  $\text{float}$  型数组;
- 声明一个大小为 8 的指向  $\text{float}$  型数的指针数组;
- 声明一个指针, 它指向大小为 8 的  $\text{float}$  型数组;
- 声明一个指针, 它指向大小为 8 的指向  $\text{float}$  型数的指针数组;
- 声明一个函数, 它返回一个  $\text{float}$  型数;
- 声明一个函数, 它返回一个指向  $\text{float}$  型数的指针;
- 声明一个指针, 它指向一个函数, 该函数返回一个  $\text{float}$  型数;
- 声明一个指针, 它指向一个函数, 该函数返回一个指向  $\text{float}$  型数的指针。

## 习 题

7.1 写出一个函数, 该函数通过指针复制一个  $\text{double}$  型数组。

7.2 写出一个函数, 该函数通过指针在一个给定数组中寻找一个给定整数。如果找到该整数, 则返回它的地址; 否则返回空指针。

7.3 写出一个函数, 传递给该函数一个由  $n$  个指向  $\text{float}$  型数的指针组成的指针数组, 要求它返回一个新建的数组, 该数组包含那  $n$  个  $\text{float}$  型数。

7.4 使用黎曼和函数求一个函数的积分, 使用公式:

$$\int_a^b f(x) dx = \sum_{j=1}^n f(a + jh)h$$

7.5 写出一个函数, 该函数使用给定的公差, 返回一个给定函数在给定点  $x$  处的数值微分。



使用公式：

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h}$$

7.6 写出一个函数，传递给它一个大小为  $n$  的、指向 `float` 型数的指针数组，要求该函数返回一个指针，指向那  $n$  个 `float` 型数中的最大值。

7.7 写出一个函数，传递给它一个大小为  $n$  的、指向 `float` 型数的指针数组，要求该函数返回一个新建的数组，该数组中以相反的顺序存放那  $n$  个 `float` 型数。

```
float * mirror (float * p [], int n)
```

7.8 写出以下函数，返回  $s$  要指向空字符 ‘\0’ 所必须增加的字节数。

```
unsigned len (const char * s)
```

7.9 写出以下函数，复制  $s2$  的前  $n$  个字节到  $s1$  前面的字节中，其中  $n$  是  $s2$  要指向空字符 ‘\0’ 所必须增加的字节数。

```
void cpy (char * s1, const char * s2)
```

7.10 写出以下函数，复制  $s2$  的前  $n$  个字节到  $s1$  中从第一个空字符 ‘\0’ 所在位置开始的字节中，其中  $n$  是  $s2$  要指向空字符 ‘\0’ 所必须增加的字节数。

```
void cat (char * s1, const char * s2)
```

7.11 写出以下函数，比较  $s2$  开始的至多  $n$  个字节与  $s1$  开始的相应字节，其中  $n$  是  $s2$  要指向空字符 ‘\0’ 所必须增加的字节数。如果所有的  $n$  个字节都匹配，函数返回 0；否则，根据第一次错配时  $s1$  中的字节是小于或大于  $s2$  中的字节，函数返回 -1 或 1。

```
int cmp (char * s1, char * s2)
```

7.12 写出以下函数，在  $s$  开始的  $n$  个字节中查找字符  $c$ ，其中  $n$  是  $s$  要指向空字符 ‘\0’ 所必须增加的字节数。如果找到该字符，返回该字符的指针；否则返回空指针。

```
char * chr (char * s, char c)
```

7.13 写出以下函数，返回数组  $p$  的前  $n$  个指针所指向的 `float` 型数的和。

```
float sum (float * p [], int n)
```

7.14 写出以下函数，修改数组  $p$  的前  $n$  个指针所指向的 `float` 型的负数的符号。

```
void abs (float * p [], int n)
```

7.15 写出以下函数，通过重新排列指针，实现对数组  $p$  的前  $n$  个指针所指向的 `float` 型数的间接排序。

```
void sort (float * p [], int n)
```

7.16 使用数组指针实现间接选择排序（参见习题 6.19 和例 7.17）。

7.17 实现间接插入排序 (参见习题 6.18 和例 7.17)。

7.18 实现间接完全打乱 (参见习题 6.29)。

7.19 改写 `sum()` 函数 (例 7.18), 使它可用于返回类型为 `float` 型的函数, 然后用 `sqrt()` 函数 (在 `<math.h>` 中定义) 和倒数函数进行测试。

7.20 将 `riemann()` 函数 (习题 7.4) 应用于以下定义于 `<math.h>` 中的函数:

a. `sqrt()`, 区间为  $[1, 4]$

b. `cos()`, 区间为  $[0, \pi/2]$

c. `exp()`, 区间为  $[0, 1]$

d. `log()`, 区间为  $[1, e]$

7.21 将 `derivative()` 函数 (习题 7.5) 应用于以下定义于 `<math.h>` 中的函数:

a. `sqrt()`, 在点  $x = 4$  处

b. `cos()`, 在点  $x = \pi/6$  处

c. `exp()`, 在点  $x = 0$  处

d. `log()`, 在点  $x = 1$  处

7.22 写出以下函数, 返回  $f(1), f(2), \dots, f(n)$  的值 (参见例 7.18)。

```
int product (int (*pf) (int k), int n)
```

7.23 实现二分法来解方程。使用以下函数:

```
double root (double (*pf) (double x), double a, double b, int n)
```

这里, `pf` 指向一个函数 `f`, `f` 用来定义要求解的方程  $f(x) = 0$ , `a` 和 `b` 是未知解  $x$  的上下界 (即  $a \leq x \leq b$ ), `n` 是循环次数。例如, 若  $f(x) = x^2 - 2$ , 则 `root(f, 1, 2, 100)` 将返回 `1.414213562373095` ( $=\sqrt{2}$ ), 即为方程  $x^2 = 2$  的解。二分法的原理是反复地把区间分为两等分, 然后用其中含有解的一半来代替该区间。它通过检查  $f(a)$  和  $f(b)$  的符号来判断解是否在区间  $[a, b]$  中。

7.24 实现梯形法则求函数的积分, 使用下面的函数:

```
double trap (double (*pf) (double x), double a, double b, int n)
```

其中, `pf` 指向求积分的函数, `a` 和 `b` 是积分区间  $[a, b]$  的上下界, `n` 是所用的子区间的个数。例如, 调用 `trap(square, 1, 2, 100)` 将返回 `1.41421`。梯形法则返回近似于  $f$  图形面积的  $n$  个梯形面积的和, 例如, 如果  $n = 5$ , 则梯形法则将返回下面的值:

$$\frac{h}{2} [f(a) + 2f(a+h) + 2f(a+2h) + 2f(a+3h) + 2f(a+4h) + f(b)]$$

其中,  $h = (b - a) / 5$ , 是每个梯形的宽度。

## 复习题答案

- 7.1 对变量  $x$  使用地址运算符  $\&$ :  $\&x$ 。
- 7.2 对指针  $p$  使用引用运算符  $*$ :  $*p$ 。
- 7.3 声明  $\text{int } n1 = n$ ; 将  $n1$  定义为  $n$  的副本; 它是一个与  $n$  具有相同值的独立变量。  
声明  $\text{int} \& n2 = n$ ; 将  $n2$  定义为  $n$  的同义字; 它跟  $n$  是同一个变量, 具有相同的地址。
- 7.4 声明  $\text{int} \& r = n$ ; 将  $r$  声明为  $\text{int}$  型变量  $n$  的一个引用 (别名)。赋值语句  $p = \&n$ ; 将  $n$  的地址赋给指针  $p$ 。
- 7.5 声明  $\text{int} * q = p$ ; 将  $q$  声明为一个指针 (内存地址), 它与  $p$  指向同一个  $\text{int}$  型数。赋值语句  $n = *p$ ; 将  $p$  指向的  $\text{int}$  型数赋给  $n$ 。
- 7.6 a. 正确。因为  $\&x == x$  且  $\&y == y$ ,  $\&x$  和  $\&y$  分别是  $x$  和  $y$  的同义字, 所以如果 ( $x == y$ ), 则它们具有相同的值。  
b. 错误。不同的对象可以有相同的名字, 但它们必须有不同的地址。
- 7.7 a. “虚悬指针”是一个未被初始化的指针, 由于它可能指向未分配的或不可访问的内存, 所以是很危险的。  
b. 如果对一个未分配的指针取值, 将会改变某些不可知的变量的值。如果对一个指向不可访问的内存的指针取值, 程序可能崩溃 (即突然终止)。  
c. 在声明指针时对它初始化。
- 7.8 一个常量不能有引用, 因为它的地址是不能访问的。
- 7.9 引用运算符  $\&$  不能应用于常量。
- 7.10 变量  $p$  的类型为  $\text{char}$ , 而表达式  $\&c$  的类型是指向  $\text{char}$  型的指针, 为将  $p$  初始化为  $\&c$ ,  $p$  的类型必须声明为  $\text{char} *$ 。
- 7.11 该声明是无效的, 因为表达式  $\&\&n$  是非法的。引用运算符  $\&$  只能用于对象 (变量和类的实例)。但是  $\&n$  不是一个对象, 它只是一个引用, 引用是没有地址的, 所以  $\&\&n$  不存在。
- 7.12 “静态绑定”是在编译时分配内存, 正如数组的声明:  

```
double a [400];
```

  
“动态绑定”是在运行时分配内存, 使用  $\text{new}$  运算符, 例如:  

```
double * p;  
p = new double [400];
```
- 7.13 变量  $p$  的类型为  $\text{char} *$ , 而表达式  $c$  的类型为  $\text{char}$ 。要将  $p$  初始化为  $c$ ,  $p$  与  $c$  应该具有相同的类型: 都为  $\text{char}$  或  $\text{char} *$ 。
- 7.14 惟一的错误是数组名  $a$  是一个指针常量, 所以它不能递增, 可以这样修改:  

```
short a [32];
```

```
short * p = a;
for (int i = 0; i < 32; i++)
    *p++ = i * i;
```

7.15 a. m = 46

b. n = 44

c. &m = 0x3fffd00

d. \*p = 46

e. r = 46

f. \*q = 46

7.16 将下面的变量按照可变左值、不变左值和非左值分类：

a. 可变左值

b. 非左值

c. 不变左值

d. 不变左值

e. 可变左值

f. 不变左值

g. 如果返回值为非局部引用，则为可变左值；否则为非左值

h. 可变左值

i. 可变左值

j. 当 p 指向一个常量时为可变左值，其他情况 \*p 是不变左值

k. 可变左值

l. 不变左值

7.17 指针 p 和 q 具有不同类型：p 指向 float 型的，而 q 指向 short 型。将一种类型的指针所指向的地址赋给另一种类型的指针是错误的。

7.18 错误在于将两个指针相加。

7.19 测试空指针并看它是否为空。特别要注意，不能对空指针取值。

7.20 p 是一个指针，它指向一个指向 double 型数的指针的指针的指针，可以用来表示四维数组。

7.21 p 的值即 x 的地址：0x3fffd1c。q 的值取决于 sizeof(double)。如果 double 类型的对象占用 8 个字节，则将偏移量 8 (5) = 40 与 p 相加，可得 q 的十六进制地址 0x3fffd44。

7.22 这些代码中非法的表达式为 p + q 和 n - q。

7.23 数组名是一个变量，其中包含数组第一个元素的地址，这个地址不能修改，所以数组名实际上是一个指针常量。

7.24 在下面的代码中，将数组中的所有元素相加，指针 p 每次加 1 都指向下一个元素：

```
const SIZE = 3;
short a [SIZE] = {22, 33, 44}
short * end = a + SIZE; //adds SIZE * sizeof(short) = 6 to a
for (short * p = a; p < end; p++)
```

```
sum += *p;
```

- 7.25 `a[i]` 的值是通过下标运算符 `[]` 返回的, 它存放在表达式 `a+i` 所计算出的地址中。在该表达式中, `a` 是指向其类型 `T` 的指针, `i` 是一个 `int` 型数, 所以将偏移量 `i * sizeof(T)` 与地址 `a` 相加。依次类推, 可以将表达式 `i+a` 同样应用于 `i[a]`。
- 7.26 `double * f();` 声明了 `f` 是一个函数, 返回一个指向 `float` 型的指针; `double (* f)();` 声明了 `f` 是一个指针, 指向一个返回 `double` 型数的函数。
- 7.27
- a. `float a[8];`
  - b. `float * a(8);`
  - c. `float (* a)[8];`
  - d. `float * (* a)[8];`
  - e. `float f();`
  - f. `float * f();`
  - g. `float (* f)();`
  - h. `float * (* f)();`

## 习题答案

- 7.1 `copy()` 函数使用 `new` 运算符给一个大小为 `n` 的 `double` 型数组分配内存, 指针 `p` 包含这个新数组的第一个元素的地址, 所以它可以同数组名一样使用, 如 `p[i]`。在将数组 `a` 中的元素复制到新数组中后, `p` 由函数返回。

```
double* copy(double a[], int n)
{ double* p = new double[n];
  for (int i = 0; i < n; i++)
    p[i] = a[i];
  return p;
}
```

```
void print(double [], int);
```

```
int main()
{ double a[8] = {22.2, 33.3, 44.4, 55.5, 66.6, 77.7, 88.8, 99.9};
  print(a, 8);
  double* b = copy(a, 8);
  a[2] = a[4] = 11.1;
  print(a, 8);
  print(b, 8);
}
```

22.2,	33.3,	44.4,	55.5,	66.6,	77.7,	88.8,	99.9
22.2,	33.3,	11.1,	55.5,	11.1,	77.7,	88.7,	99.9
22.2,	33.3,	44.4,	55.5,	66.6,	77.7,	88.8,	99.9

在运行中, 初始化 `a` 为一个大小为 8 的 `double` 型数组, 使用 `print()` 函数检查 `a` 的内容。调用 `copy()` 函数并将其返回值赋给指针 `b`, 然后将 `b` 用做新数组的名字。在打印 `b` 之前, 修改 `a` 中两个元素的值, 以检查 `b` 和 `a` 不是同一个数组, 最后调用的两个 `print()` 验证了这一点。

7.2 使用 `for` 循环来遍历该数组。如果在 `a[i]` 找到目标, 则返回它的地址 `&a[i]`; 否则返回空指针。

```
int* location(int a[], int n, int target)
{ for (int i = 0; i < n; i++)
    if (a[i] == target) return &a[i];
  return NULL;
}
```

下面的测试程序调用该函数, 然后将函数返回的地址存在指针 `p` 中。如果 `p` 非零 (即不是空指针), 则打印 `p` 和 `p` 所指向的 `int` 型数。

```
int main()
{ int a[8] = {22, 33, 44, 55, 66, 77, 88, 99}, * p, n;
  do
  { cin >> n;
    if (p = location(a, 8, n)) cout << p << ", " << *p << endl;
    else cout << n << " was not found.\n";
  } while (n > 0);
}
```

```
44
0x3fffc0d4, 44
50
50 was not found.
99
0x3fffc0d8, 99
90
90 was not found.
0
0 was not found.
```

7.3 使用 `for` 循环来遍历该数组, 直到 `p` 指向目标。

```
float* duplicate(float* p[], int n)
{ float* const b = new float[n];
  for (int i = 0; i < n; i++)
    b[i] = *p[i];
  return b;
}
```

```
void print(float [], int);
void print(float* [], int);
```

```
int main()
```

```

{ float a[8] = {44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5};
  print(a, 8);
  float* p[8];
  for (int i = 0; i < 8; i++)
    p[i] = &a[i]; //p[i]指向a[i]
  print(p, 8);
  float* const b = duplicate(p, 8);
  print(b, 8);
}

```

```

44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5
44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5
44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5

```

7.4 这个名为 `riemann()` 的函数与例 7.18 中的 `sum()` 函数类似。它的第一个参数是一个指向函数的指针，该函数有一个 `double` 型的参数并返回一个 `double` 型值，在测试运行时，将这个指针传递给（指向）`cube()` 函数。其余的三个参数是：计算积分的区间  $[a, b]$  的上下界  $a$  和  $b$ ，在求和时使用的子区间的个数  $n$ 。实际的黎曼和是这些子区间上的  $n$  个梯形的面积的和，子区间的高度是由被积分的函数给定的。

```

double riemann(double (*)(double), double, double, int);
double cube(double);

int main()
{ cout << riemann(cube, 0, 2, 10) << endl;
  cout << riemann(cube, 0, 2, 100) << endl;
  cout << riemann(cube, 0, 2, 1000) << endl;
  cout << riemann(cube, 0, 2, 10000) << endl;
}

// 返回 [f(a)*h+f(a+h)*h+f(a+2h)*h+...+f(b-h)*h]
// 此时 h=(b-a)/n
double riemann(double (*pf)(double t), double a, double b, int n)
{ double s = 0, h = (b-a)/n, x;
  int i;
  for (x = a, i = 0; i < n; x += h, i++)
    s += (*pf)(x);
  return s*h;
}

double cube(double t)
{ return t*t*t;
}

```

```

3.24
3.9204
3.992
3.9992

```

测试运行时，在区间  $[0, 2]$  上求函数  $y = x^3$  的积分。通过计算，积分值为 4.0。调

用 `riemann (cube, 0, 2, 10)` 使用 10 个子区间来求积分的近似值, 结果为 3.24。调用 `riemann (cube, 0, 2, 100)` 使用 100 个子区间来求积分的近似值, 结果为 3.9204。随着  $n$  的增加和越来越接近于极限 4.0。当使用 10000 个子区间时, 黎曼和是 3.9992。注意 `riemann ()` 函数与例 7.18 中的 `sum ()` 函数惟一重要的区别在于: 和在返回之前与子区间的宽度  $h$  相乘。

- 7.5 `derivative ()` 函数与例 7.18 中的 `sum ()` 函数非常类似, 除了它是使用数值微分。该函数有三个参数: 一个指向函数  $f$  的指针、 $x$  值和公差  $h$ 。在测试运行时, 将它传递给 (指向) 函数 `cube ()` 和 `sqrt ()` 函数。

```
#include <iostream>
#include <cmath>
using namespace std;
double derivative(double (*)(double), double, double);
double cube(double);

int main()
{ cout << derivative(cube, 1, 0.1) << endl;
  cout << derivative(cube, 1, 0.01) << endl;
  cout << derivative(cube, 1, 0.001) << endl;
  cout << derivative(sqrt, 1, 0.1) << endl;
  cout << derivative(sqrt, 1, 0.01) << endl;
  cout << derivative(sqrt, 1, 0.001) << endl;
}

// 返回导数 f'(x) 的一个近似值
double derivative(double (*pf)(double t), double x, double h)
{ return ((*pf)(x+h) - (*pf)(x-h))/(2*h);
}

double cube(double t)
{ return t*t*t;
}
```

```
3.11
3.0001
3
0.500628
0.500006
0.5
```

`cube ()` 函数  $x^3$  的微分为  $3x^2$ , 当  $x=1$  时其值为 3, 所以, 对于较小的  $h$ , 其数值微分接近于 3。与此类似, `sqrt ()` 函数的微分  $\sqrt{x}$  是  $1/(2\sqrt{x})$ , 在  $x=1$  时其值为  $1/2$ , 所以, 对于较小的  $h$ , 其数值微分接近于 0.5。

- 7.6 使用指针 `pmax` 指向最大的 `float` 数, 它被初始化为 `p[0]`, 而 `p[0]` 指向第一个 `float` 数, 然后在 `for` 循环中, `p[i]` 所指向的 `float` 数与 `pmax` 所指向的 `float` 数相比较, 并且将 `pmax` 的值更新使它指向其中较大的 `float` 数。所以当循环终止时, `pmax` 指向最大



的 float 数。

```
float* max(float* p[], int n)
{ float* pmax = p[0];
  for (int i = 1; i < n; i++)
    if (*p[i] > *pmax) pmax = p[i];
  return pmax;
}

void print(float [], int);
void print(float* [], int);

int main()
{ float a[8] = {44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5};
  print(a, 8);
  float* p[8];
  for (int i = 0; i < 8; i++)
    p[i] = &a[i]; // P[i] 指向 a[i]
  print(p, 8);
  float* m = max(p, 8);
  cout << m << ", " << *m << endl;
}
```

```
44.7, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5
44.7, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5
0x3ffffcd4, 99.9
```

这里使用了两个重载的 `print()` 函数：一个用来打印指针数组，另一个用来打印这些指针所指向的 float 数。在初始化并打印数组 `a` 之后，定义数组 `p` 并初始化 `p` 的元素为指向数组 `a` 的元素。调用 `print(p, 8)` 检验 `p` 提供了间接访问 `a` 的方法。最后，声明指针 `m` 并用 `max()` 函数返回的地址对 `m` 进行初始化。最后一个输出语句证明 `m` 确实真正指向 `p` 所访问的那些 float 数中最大的一个。

在线访问 [projectEuclid.net](http://projectEuclid.net)，可得到习题 7.7~7.24 的答案。

# 第8章 字符串

## 8.1 介绍

字符串是一些字符的序列，在内存中以空字符‘\0’结尾。字符串可以通过类型为 char \* 型的变量（指向 char 的指针）来访问。例如，如果 s 的类型为 char \*，则

```
cout << s << endl;
```

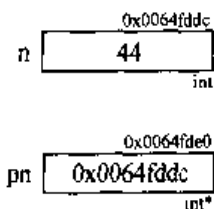
将打印从地址 s 开始到第一个空字符之间的内存空间中存放的所有字符。

C 的头文件 <cstring> 中提供了非常丰富的函数，可以对字符串进行操作。例如，调用 strlen (s) 将返回字符串 s 中的字符个数。这些函数中都将字符串参数的类型声明为指向 char 的指针，所以在学习这些字符串操作之前，需要复习指针的知识（参见第 7.3 节）。

## 8.2 复习指针

指针是一个内存地址。例如，以下的声明定义了一个值为 44 的 int 型数 n，还有包含 n 的地址的指针 pn（如图 8.1 所示）。

```
int n = 44;  
int * pn = &n;
```



如果将内存想像为具有十六进制地址的一连串的字节，则可以画出 n 和 pn，如图 8.2 所示。

图 8.2 中表示 n 存放在地址 64fddc 中，并且 pn 存放了地址 64fdd0，变量 n 包含值 44 而变量 pn 包含地址值 64fddc，pn 的值是 n 的地址，这种关系通常用一个简单的图表示，图 8.3 中显示了两个矩形，一个标为 n，另一个标为 pn，矩形代表内存单元。变量 pn 指向变量 n，可以使用取值运算符 \* 通过指针 pn 来访问 n。例如，语句：

```
*pn = 77;
```

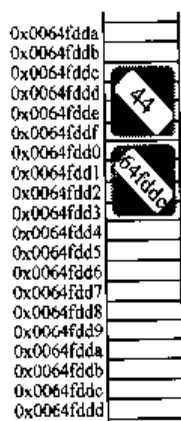
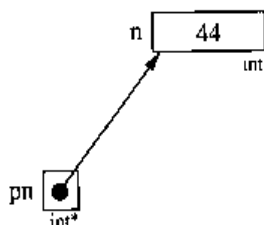
将把 n 的值改为 77。

可以用多个指针指向同一个对象。例如：

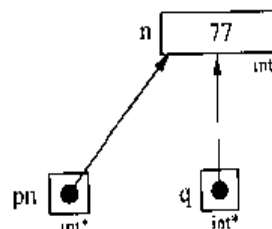
```
float * q = &x;
```

现在 \*pn、\*q 和 x 是同一个对象的名字，该对象的地址为 64fddc 且当前值为 77，如

图 8.4 所示。其中,  $q$  存放在地址 64fde4 中,  $q$  的值是  $n$  的地址 64fddc。

图 8.2  $n$  与  $pn$  在内存中的位置图 8.3  $pn$  指向  $n$ 图 8.4  $n$ 、 $pn$ 、 $q$  在内存中的位置

下例是这些定义在装有 Pentium III 处理器的 Windows 工作站上的 Metrowerks Code Warrior C++ 软件中运行的情况。如图 8.5 所示, 内存是以升序分配的。第一个对象  $n$  从地址 65fcc8 开始存放, 占用字节 65fcc8 ~ 65fccb。第二个对象  $pn$  存放在地址 65fccc 中, 第三个对象  $q$  存放在地址 65fcd0 中。

图 8.5  $pn$ 、 $q$  指向一个目标

### 例 8.1 跟踪指针

下例与例 7.5 的运行结果相同:

```
int main ()
{ int n=44;    // n保留整数 44
  cout << "int n=44;    // n保留整数 44: \n";
  cout << "\t\t\t n = " << n << endl;
  cout << "\t\t\t &n = " << &n << endl;
  int * pn=&n;  // pn保留 n的地址
  cout << "int * pn=&n;  // pn保留 n的地址: \n";
  cout << "\t\t\t n = " << n << endl;
  cout << "\t\t\t &n = " << &n << endl;
  cout << "\t\t\t pn = " << pn << endl;
  cout << "\t\t\t &pn = " << &pn << endl;
  cout << "\t\t\t *pn = " << *pn << endl;
  *pn = 77;    // 改变 n的值为 77
  cout << "*pn = 77;    // 改变 n的值为 77: \n";
  cout << "\t\t\t n = " << n << endl;
  cout << "\t\t\t &n = " << &n << endl;
  cout << "\t\t\t pn = " << pn << endl;
  cout << "\t\t\t &pn = " << &pn << endl;
  cout << "\t\t\t *pn = " << *pn << endl;
  int * q=&n;  // q也保留 n的地址
  cout << "int * q=&n;  // q也保留 n的地址: \n";
  cout << "\t\t\t n = " << n << endl;
  cout << "\t\t\t &n = " << &n << endl;
```

```

cout << "\t\t\t pn = " << pn << endl;
cout << "\t\t\t &pn = " << &pn << endl;
cout << "\t\t\t *pn = " << *pn << endl;
cout << "\t\t\t q = " << q << endl;
cout << "\t\t\t &q = " << &q << endl;
cout << "\t\t\t *q = " << *q << endl;

```

```

int * pn = &n; // pn 存放 n 的地址:
                n = 44
                &n = 0x0065fcc8
                pn = 0x0065fcc8
                &pn = 0x0065fccc
                *pn = 44
*pn = 77; // 将 n 的值改为 77:
                n = 77
                &n = 0x0065fcc8
                pn = 0x0065fcc8
                &pn = 0x0065fccc
                *pn = 77
int * q = &n; // pn 存放 n 的地址:
                n = 77
                &n = 0x0065fcc8
                pn = 0x0065fcc8
                &pn = 0x0065fccc
                *pn = 77
                q = 0x0065fcc8
                &q = 0x0065fccc
                *q = 77

```

如果  $p$  为一个指针，则语句 `cout << p` 将始终打印  $p$  所指向的对象的值，并且语句 `cout << &p` 将始终打印在  $p$  中存放的地址。对第二条规则，有一个非常重要的例外，就是当  $p$  的类型被声明为 `char *` 时。

### 8.3 字符串

在 C++ 中，字符串是一个字符数组，具有以下的重要特征：

- 一个额外的字符被追加到数组的末尾，且它的值被设置为空字符 `'\0'`，这意味着在数组中字符的总个数总是比串长多 1。
- 可以用一个字符串值来初始化一个字符串，如下所示：

```
char str [] = "Bjarne";
```

注意这个数组有 7 个元素：`'B'`，`'j'`，`'a'`，`'r'`，`'n'`，`'e'` 和 `'\0'`。

- 整个字符串可以作为单个对象输出，如下所示：

```
cout << str;
```

系统将字符从 `str` 复制到 `cout`，直到遇到空字符 `'\0'`。

- 整个字符串可以作为单个对象输入，如下所示：

```
cin >> buffer;
```

系统将字符从 cin 复制到 buffer，直到遇到空字符 '\0'。用户必须确保所定义的缓冲区 buffer 可以有足够的空间来存放输入的字符。

- 在头文件 <cstring> 中声明的函数可以用来对字符串进行操作，包括：字符串长度函数 strlen()，字符串复制函数 strcpy() 和 strncpy()，字符串连接函数 strcat() 和 strncat()，字符串比较函数 strcmp() 和 strncmp()，取出标记函数 strtok()。这些函数将在第 8.8 节中描述。

### 例 8.2 字符串是以空字符终止的

下面的这个小例子说明追加到字符串中的空字符 '\0'。

```
int main ()
{ char s [] = "ABCD";
  for (int i = 0; i < 5; i++)
    cout << "s [" << i << "] = " << s [i] << " \n";
```

```
s [0] = 'A'
s [1] = 'B'
s [2] = 'C'
s [3] = 'D'
s [4] = '\0'
```

当空字符被送到 cout，将什么也不打印——即使是一个空白，这点可以通过紧挨着该字符的前后各打印一个省略号来观察。

## 8.4 字符串的输入输出

在 C++ 程序中可以有几种方法来完成字符串的输入和输出操作。一种方法是使用标准的 C++ string 运算符，这里主要介绍其他的方法。

### 例 8.3 普通的字符串输入输出

本例读入并处理一个 79 个字符的字符串缓冲区：

```
int main ()
{ char word [80];
  do
  { cin >> word;
    if (*word) cout << "\t\t" << word << "\n";
  } while (*word);
}
```

```
Today's date is March 12, 2000.
    "Today's"
    "date"
```

```

    " is"
    " March"
    " 12,"
    "2000."
    Tomorrow is Monday.
    "Tomorrow"
    "is"
    "Monday."
    ^Z

```

程序运行时，while 循环重复执行 10 次：每次处理输入的是一个单词（包括终止循环的 Ctrl+Z）。输入流 cin 中的每个单词在输出流 cout 中被重复打印出来。注意当输入流到行尾时，输出流才开始工作。

打印每个字符串时，在它的两侧各打印一个双引号，该字符必须在字符串文字中用字符对 \ 指明。

表达式 \*word 控制循环。它是字符串中的初始字符。当字符串 word 中包含一个串长大于 0 的字符串时，它的值为非零（即真）。串长为 0 的字符串称为空串，包含有一个空字符 '\0'，输入 Ctrl+Z+Enter 将文件结束符传递给 cin，并将空串装入 word，设置 \*word（与 word[0] 相同）为 '\0' 且终止循环。输出的最后一行表示仅打印 Ctrl+Z，显示为 ^Z。

在按下 Ctrl+Z 后，应按两次 Enter 键。

注意标点符号（省略号，逗号，句号等）是包括在字符串中的，但空白符（空格，制表符，换行符等）不包括在字符串中。

例 8.3 中的 do 循环可以用下面的代码代替：

```

cin >> word;
while (*word)
{ cout << "\"t\"" << word << "\"\n";
  cin >> word;
}

```

当按下 Ctrl+Z 时，调用 cin >> word 将字符串赋给 word。

例 8.3 和例 8.1 说明了一个非常重要的特性：当输出运算符和类型为 char\* 的指针一起使用时，其作用与输出运算符和其他类型的指针一起使用时不同。此时，输出运算符输出该指针所指向的整个字符串，但是当和其他类型的指针一起使用时，输出运算符仅仅输出指针的地址。

## 8.5 cin 成员函数

输入流对象 cin 包含输入函数：cin.getline(), cin.get(), cin.ignore(), cin.putback(), cin.peek()。每个函数名中都带有前缀“cin.”，因为它们是对对象 cin 的“成员函数”。

调用 cin.getline(str, n) 将前面的 n 个字符读入 str，而忽略剩余的字符。

### 例 8.4 带有两个参数的 cin.getline

本例逐行重复打印输入：

```
int main ()
{ char line [80];
  do
  { cin.getline (line, 80);
    if (*line) cout << "\t [" << line << "]" \n";
  } while (*line);
}
```

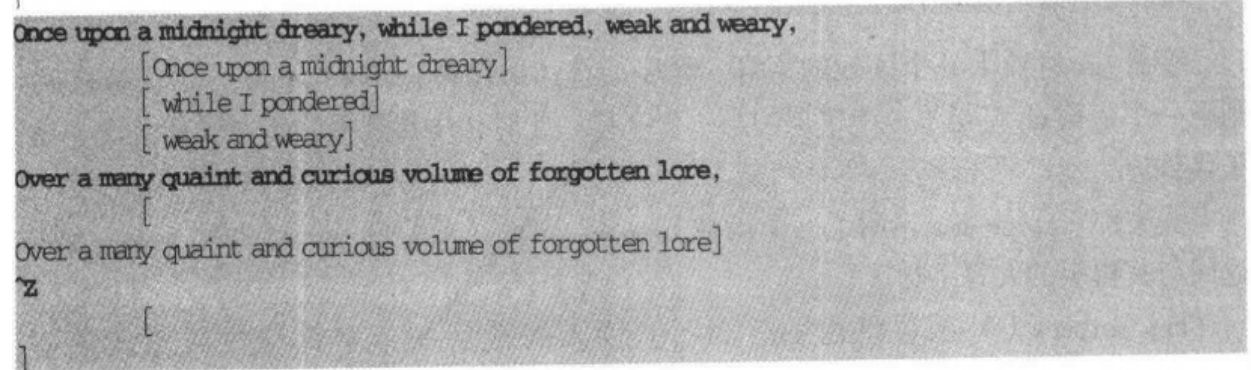
注意，只有当 line 中包含一个非空的字符串时，条件 (\*line) 的值才为真，因为只有这样 line [0] 的值才与空字符 (ASCII 码值为 0) 不同。

调用 cin.getline(str,n,ch) 将定界符 ch 前面的所有字符读入 str。如果指定字符 ch 是换行符 '\n'，则它与 cin.getline(str,n) 等价。下例说明了这一点，其中定界符为逗号“,”。

### 例 8.5 带有三个参数的 cin.getline () 函数

本例逐句重复打印输入：

```
int main ()
{ char clause [80];
  do
  { cin.getline (clause, 80, ',');
    if (*clause) cout << "\t [" << clause << "]" \n";
  } while (*clause);
}
```



```
Once upon a midnight dreary, while I pondered, weak and weary,
    [Once upon a midnight dreary]
    [ while I pondered]
    [ weak and weary]
Over a many quaint and curious volume of forgotten lore,
    [
Over a many quaint and curious volume of forgotten lore]
^Z
    [
]
```

注意跟在“weary”后面的不可见的行结束符，是作为下一个输入行的第一个字符存放的。由于使用了逗号作为定界符，所以行结束符是作为普通字符处理的。

cin.get () 函数用于逐字符输入，调用 cin.get (ch) 将输入流 cin 中的下一个字符复制到变量 ch 中，并返回 1，如果测试到文件结束则返回 0。

### 例 8.6 cin.get () 函数

本例计算字母'e'在输入流中出现的个数，循环反复执行至 cin.get (ch) 函数成功地将字符读入 ch。

```
int main ()
{ char ch;
  int count = 0;
  while (cin.get (ch))
    if (ch == 'e') ++count;
  cout << count << " e's were counted. \n";
}
```

```
Once upon a midnight dreary, while I pondered, weak and weary,
Over many a quaint and curious volume of forgotten lore,
^Z
11 e's were counted.
```

与 `get` 相反的是 `put`。函数 `cout.put ()` 用于逐字符地写输出流 `cout`，如下例所示。

### 例 8.7 `cout.put ()` 函数

本例重复打印输入，将每个单词的开头字母大写：

```
int main ()
{ char ch, pre = '\0';
  while (cin.get (ch))
    { if (pre == ' ' || pre == '\n') cout.put (char (toupper (ch)));
      else cout.put (ch);
      pre = ch;
    }
}
```

```
Fourscore and sever years ago our fathers
Fourscore And Sever Years Ago Our Fathers
Brought forth upon this continent a new nation
Brought Forth Upon This Continent A New Nation
^Z
```

变量 `pre` 存放上一个读入的字符。程序的设计思想为：如果 `pre` 是一个空格或换行符，则下一个字符 `ch` 将是下一个单词的第一个字母。此时，用与其相应的大写字母 `ch + 'A' - 'a'` 来代替 `ch`。

头文件 `<ctype.h>` 声明了一个函数 `toupper (ch)`，如果 `ch` 是一个小写字母，则该函数返回与其相应的大写字母。

`cin.putback ()` 函数将由 `cin.get ()` 读入的最后一个字符存放回输入流 `cin` 中，`cin.ignore ()` 函数读输入流中的一个和多个字符却不作处理，如例 8.8 所示。

`cin.peek ()` 函数的功能相当于联合使用 `cin.get ()` 和 `cin.putback ()` 函数。调用

```
ch = cin.peek;
```

将输入流 `cin` 中的下一个字符复制到 `char` 变量 `ch` 中，而不从输入流中删除该字符。例 8.9 说明如何使用 `peek ()` 函数来代替 `get ()` 和 `putback ()` 函数。

### 例 8.8 `cin.putback ()` 和 `cin.ignore ()` 函数

本例测试一个函数，该函数从输入流中抽取整数：



```

int nextInt ();
int main ()
{ int m = nextInt (), n = nextInt ();
  cin.ignore (80, '\n');           // 忽略输入计的剩余部分
  cout << m << " + " << n << " = " << m+n << endl;
}
int nextInt ()
{ char ch;
  int n;
  while (cin.get (ch))
    if (ch >= '0' && ch <= '9')    // 下个字符是一个数字
    { cin.putback (ch);           // 把它放回使得
      cin >> n;                   // 它可以作为一个完整的整数读入
      break;
    }
  return n;
}

```

What is 305 plus 9416?  
305 + 9416 = 9721

函数 `nextInt ()` 在 `cin` 中扫描字符，直到遇到第一个数字，在本次运行时为数字 3。由于 3 是第一个整数 305 的一部分，它被放回 `cin`，目的是可以将完整的整数 305 读入 `n`，并返回 305。

### 例 8.9 `cin.peek ()` 函数

本例中的 `nextInt ()` 函数与上例中的 `nextInt ()` 等价：

```

int nextInt ()
{ char ch;
  int n;
  while (ch = cin.peek ())
    if (ch >= '0' && ch <= '9')
    { cin >> n;
      break;
    }
    else cin.get (ch);
  return n;
}

```

表达式 `ch = cin.peek ()` 将下一个字符复制进 `ch`，并在成功后返回 1。然后，如果 `ch` 是个数字，则完整的整数被读入 `n` 并被返回。否则，将从 `cin` 中删除 `ch` 并继续循环。如果遇到文件结束符，表达式 `ch = cin.peek ()` 返回 0，然后终止循环。

## 8.6 标准的 C 字符函数

在例 8.7 中看到 `toupper ()` 函数是定义在 `<cctype>` 中的一系列字符操作函数中的一个，下表总结了这些函数：

isalnum ( )	int isalnum (int c); 当 c 为一个字母或数字字符时返回非 0; 否则返回 0。
isalpha ( )	int isalpha (int c); 当 c 为一个字母字符时返回非 0; 否则返回 0。
isctrl ( )	int isctrl (int c); 当 c 为一个控制字符时返回非 0; 否则返回 0。
isdigit ( )	int isdigit (int c); 当 c 为一个数字字符时返回非 0; 否则返回 0。
isgraph ( )	int isgraph (int c); 当 c 为一个任意非空的可打印字符时返回非 0; 否则返回 0。
islower ( )	int islower (int c); 当 c 为一个小写字母字符时返回非 0; 否则返回 0。
isprint ( )	int isprint (int c); 当 c 为一个任意可打印字符时返回非 0; 否则返回 0。
ispunct ( )	int ispunct (int c); 当 c 为一个除字母、数字和空格字符之外的任意可打印字符时返回非 0; 否则返回 0。
isspace ( )	int isspace (int c); 当 c 为任意一个空白字符时 (包括空格符 ' ', 换页符 '\f', 换行符 '\n', 回车符 '\r', 水平制表符 '\t' 和垂直制表符 '\v') 返回非 0; 否则返回 0。
isupper ( )	int isupper (int c); 当 c 为一个大写字母字符时返回非 0; 否则返回 0。
isxdigit ( )	int isxdigit (int c); 当 c 为一个 10 个数字或 12 个十六进制数的字母: 'a', 'b', 'c', 'd', 'e', 'f', 'A', 'B', 'C', 'D', 'E', 'F' 时返回非 0; 否则返回 0。
tolower ( )	int tolower (int c); 当 c 为一个大写字母字符时返回相应的小写字母; 否则返回 c。
toupper ( )	int toupper (int c); 当 c 为一个小写字母字符时返回相应的大写字母; 否则返回 c。

注意, 这些函数的参数是一个 int 型的 c, 并且返回值是一个 int 型的数, 这是因为 char 是一个整数类型。通常, 是将一个 char 型的数传递给这些函数, 并且返回值是赋给一个 char 型的数, 所以将这些函数看做字符处理函数。

## 8.7 字符串数组

一个二维数组实际上是一个元素为一维数组的一维数组, 当这些元素为字符串时, 就是一个字符串数组。

例 8.10 中声明了一个二维数组:

```
char name [5] [20];
```

这个声明分配了一百个字节, 具体安排如图 8.6 所示: 每 5 行是一个包含 20 个元素的一维数组, 它们可以看做一个字符串。可以通过 name [0], name [1], name [2], name [3], name [4] 来访问这些字符串。在例 8.10 的一次运行中, 数据的存储如图 8.7 所示。其中, 符号 Ø 表示空字符 '\0'。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0																				
1																				
2																				
3																				
4																				

图 8.6 数组 name 示例

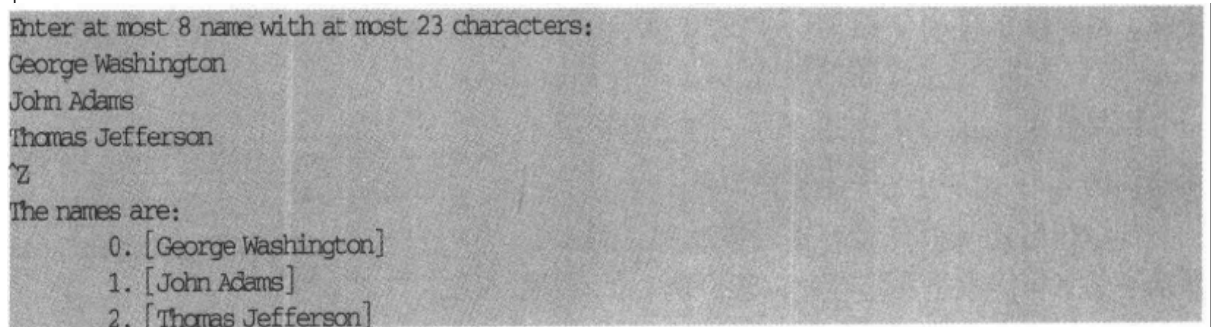
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	G	e	o	r	g	e		W	a	s	h	i	n	g	t	o	n	Ø		
1	J	o	h	n		A	d	a	m	s	Ø									
2	T	h	o	m	a	s		J	e	f	f	e	r	s	o	n	Ø			
3																				
4																				

图 8.7 数组 name 中存放的数据

**例 8.10 一个字符串数组**

本例读入一个字符串的序列，将该序列存放在一个数组中，然后打印出来。

```
int main ()
{ char name [5] [20];
  int count = 0;
  cout << "Enter at most 4 names with at most 19 characters: \n";
  while (cin.getline (name [count++], 20))
    ;
  -- count;
  cout << "The names are: \n";
  for (int i = 0; i < count; i++)
    cout << "\t" << i << ". [" << name [i] << "]" << endl;
```



```
Enter at most 8 name with at most 23 characters:
George Washington
John Adams
Thomas Jefferson
^Z
The names are:
0. [George Washington]
1. [John Adams]
2. [Thomas Jefferson]
```

注意，while 循环的所有动作都是在其控制条件内部完成的。例如：

```
cin.getline (name [count++], 20)
```

该条件调用 `cin.getline ()` 函数读入下一行字符到 `name [count]` 中，然后将 `count` 加 1。当函数成功地将一个字符串读入 `name [count]` 后返回非 0 值（即“true”）。当被告知遇到文件结束符时（通过 `<Control - D>` 或 `<Control - Z>`），`cin.getline ()` 函数操作失败，所以它返回 0，由此终止 while 循环。循环体为空，除了一个分号外什么都没有。

存储字符串更为高效的方法是声明一个指针数组：`char * name [4]`。其中，数组的每个元素的类型为 `char *`，说明每个 `name [i]` 都是一个字符串。这种声明方式在开始时不分配存储空间给字符串，而是改为将所有数据存放在字符串缓冲区中，然后设置每个 `name [i]` 等于在缓冲区中相应名字的第一个字符的地址，如例 8.11 中所做的。这种方法更为高效，因为每个 `name [i]` 只需存储字符串所需的字节数，交换条件是输入例程需要一个输入结束的标记。

**例 8.11 字符串数组**

本例说明带有标记字符 '\$' 的 `getline ()` 函数的用法，它几乎与例 8.10 相同。该例读入一个名字的序列，每行读入一个名字，使用标记 '\$' 终止，然后打印出存放在数组 `name` 中的名字。

```
int main ()
{ char buffer [80];
  cin.getline (buffer, 80, '$');
```

```

char* name[4];
name[0] = buffer;
int count = 0;
for (char* p = buffer; *p != '\0'; p++)
    if (*p == '\n')
        | *p = '\0';           // 结束名字 [count]
        | name[++count] = p+1; // 开始下一个名字
    |
cout << "The names are: \n";
for (int i = 0; i < count; i++)
    cout << "\t" << i << ". [" << name[i] << "]" << endl;
}

```

所有的输入（包括：“George Washington \n John Adams \n Thomas Jefferson \n.”）被作为单个的字符串存放在缓冲区中，for 循环使用指针 *p* 扫描缓冲区。每当 *p* 找到一个 ‘\n’ 字符，则在该字符串后追加一个空字符 ‘\0’，存放在 *name* [count] 中，然后将计数器 *count* 加 1 并将下一个字符的地址 *p*+1 存入 *name* [count]。

结果数组 *name* 如图 8.8 所示。注意这里并不需要在例 8.10 中加在名字末尾的额外字符。

如果在编译时被存储的字符串已知，则上面所描述的字符串数组更加容易处理。例 8.12 说明如何初始化一个字符串数组。

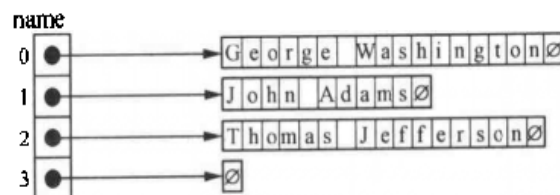


图 8.8 name 数组最终样子

### 例 8.12 初始化一个字符串数组

本例与前面的两个例子很类似，其中初始化了字符串数组 *name*，然后打印其内容：

```

int main ()
{ char* name[]
  = { "George Washington", "John Adams", "Thomas Jefferson" };
  cout << "The names are: \n";
  for (int i = 0; i < 3; i++)
      cout << "\t" << i << ". [" << name[i] << "]" << endl;
}

```

The names are:

```

0.  [George Washington]
1.  [John Adams]
2.  [Thomas Jefferson]

```

在 *name* 数组中存储的数据与例 8.11 中的相同。

## 8.8 标准 C 的字符串函数

C 的头文件 `<cstring>`，也称为字符串库，包含了许多非常有用的处理字符串的函数。例 8.13 说明了其中最简单的一个：字符串长度函数，它返回传递给它的字符串的长度。

**例 8.13 strlen () 函数**

本例是 strlen () 函数的一个简单的测试程序, 调用 strlen (s) 返回 s 中第一个空字符 '\0' 之前的所有字符的个数:

```
#include <cstring>
int main ()
{ char s [] = "ABCD EFG";
  cout << "strlen (" << s << ") = " << strlen (s) << endl;
  cout << "strlen (\" \") = " << strlen ("") << endl;
  char buffer [80];
  cout << "Enter string: "; cin >> buffer;
  cout << "strlen (" << buffer << ") = " << strlen (buffer) << endl;
}
```

在某些方面, 字符串与基本类型的对象 (即整数和实数) 相同。例如, 它们可以用同样的方法输出到 cout。但是, 字符串是结构化对象, 由较小的成分 (字符) 组成。有许多可用于基本类型对象的运算符, 如赋值运算符 (=), 比较运算符 (<, >, ==, <=, >= 和 !=), 和算术运算符 (+) 等是不能用于字符串的, 一些 C 字符串库中的函数模拟了这些运算符的作用, 在第 12 章将学习如何编写这些函数。

下例中说明了三个其他的字符串函数, 它们用于在一个给定的字符串中定位字符和子串。

**例 8.14 strchr (), strrchr () 和 strstr () 函数**

```
#include <cstring>
int main ()
{ char s [] = "The Mississippi is a long river.";
  cout << "s = \" \" << s << "\"\n";
  char * p = strchr (s, ' ');
  cout << "strchr (s, ' ') points to s [" << p - s << "] .\n";
  p = strchr (s, 's');
  cout << "strchr (s, 's') points to s [" << p - s << "] .\n";
  p = strrchr (s, 's');
  cout << "strrchr (s, 's') points to s [" << p - s << "] .\n";
  p = strstr (s, "is");
  cout << "strstr (s, \"is\") points to s [" << p - s << "] .\n";
  p = strstr (s, "isi");
  if (p == NULL) cout << "strstr (s, \"isi\") returns NULL\n";
}
```

s = "The Mississippi is a long river."  
 strchr (s, ' ') points to s [3] .  
 strchr (s, 's') points to s [6] .  
 strrchr (s, 's') points to s [17] .  
 strstr (s, "is") points to s [5] .  
 strstr (s, "isi") returns NULL.

调用 strchr (s, ' ') 返回一个指针, 指向空格字符 ' ' 在字符串 s 中第一次出现的位置, 表达式 p - s 计算在字符串中这个字符的下标 (偏移量) 3 (记住数组使用 0 阶下标, 所以, 开

始字符‘T’的下标为0)。依此类推,字符‘s’在s中第一次出现在下标6处。

调用 `strchr(s, 's')` 返回一个指针, 指向字符‘s’在字符串s中的最后一次出现的位置, 是 `s[17]`。

调用 `strstr(s, "is")` 返回一个指针, 指向子串“is”在字符串s中的第一次出现的位置, 是 `s[5]`。调用 `strstr(s, "isi")` 返回一个空指针, 因为“isi”没有在字符串s中出现。

对字符串来说, 有两个函数可以模仿赋值运算符的作用, 它们是: `strcpy()` 和 `strncpy()`。调用 `strcpy(s1, s2)` 将字符串s2复制到字符串s1中, 调用 `strncpy(s1, s2, n)` 将字符串s2的前n个字符复制到字符串s1中。两个函数都返回s1, 如以下的两个例子所示。

### 例 8.15 strcpy () 函数

本例跟踪函数 `strcpy(s1, s2)` 的调用过程:

```
#include <cstring>
#include <iostream>
int main ()
{ char s1 [] = "ABCDEFGG";
  char s2 [] = "XYZ";
  cout << "Before strcpy (s1, s2): \n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen(s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen(s2) << endl;
  strcpy(s1, s2);
  cout << "After strcpy (s1, s2): \n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen(s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen(s2) << endl;
}
```

```
Before strcpy (s1, s2):
  s1 = [ABCDEFGG], length=7
  s2 = [XYZ], length=3
After strcpy (s1, s2):
  s1 = [XYZ], length=3
  s2 = [XYZ], length=3
```

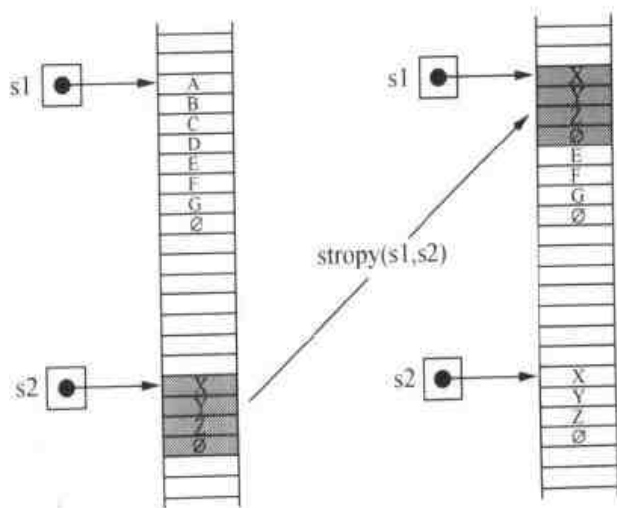


图 8.9 字符串复制结果

在s2复制到s1中之后, 就不能区分这两个字符串了; 它们都包含3个字符XYZ。 `strcpy(s1, s2)` 的结果如图8.9所示。由于s2的长度为3, `strcpy(s1, s2)` 复制4个字节(包括空字符, 表示为Φ), 覆盖了s1的前4个字节, 使s1的长度变为3。

注意, `strcpy(s1, s2)` 创建了字符串s2的一个副本, 结果中的两个字符串是不同的字符串, 改变其中的一个字符串不会影响另一个。

### 例 8.16 strncpy () 函数

本例跟踪函数 `strncpy(s1, s2)` 的调用过程:

```
int main ()
{ char s1 [] = "ABCDEFGG";
  char s2 [] = "XYZ";
  cout << "Before strncpy (s1, s2, 2): \n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen (s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen (s2) << endl;
  strncpy (s1, s2, 2);
  cout << "After strncpy (s1, s2, 2): \n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen (s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen (s2) << endl;
}
```

```
Before strncpy (s1, s2, 2):
  s1 = [ABCDEFGG], length=7
  s2 = [XYZ], length=3
After strncpy (s1, s2, 2):
  s1 = [XYCDEFGG], length=7
  s2 = [XYZ], length=3
```

调用 `strncpy (s1, s2, 2)` 用 XY 替换 s1 的前 2 个字符，而保留 s1 中剩余的字符。`strncpy (s1, s2, 2)` 的结果如图 8.10 所示。由于 s2 的长度为 3，`strncpy (s1, s2, 2)` 复制 2 字节（不包括空字符 `\0`），覆盖 s1 的前两个字符，这并不影响 s1 的长度 7。

如果  $n < \text{strlen} (s2)$ ，如上例中所示，则 `strncpy (s1, s2, n)` 只把 s2 的前 n 个字符复制到 s1 的开始。但是，如果  $n \geq \text{strlen} (s2)$ ，则 `strncpy (s1, s2, n)` 的结果与 `strcpy (s1, s2)` 相同：它使 s1 成为长度与 s2 相同的 s2 的副本。

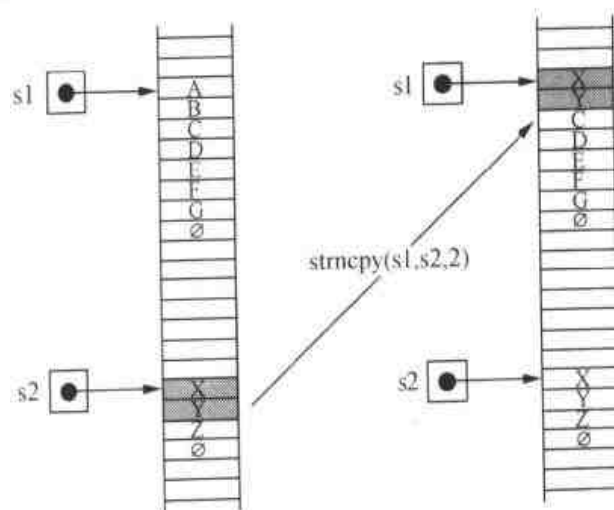


图 8.10 strncpy 示例

### 例 8.17 字符串连接函数 strcat ()

本例跟踪调用函数 `strncpy (s1, s2)`，将字符串 s2 追加到 s1 的末尾的过程：

```
int main ()
{ char s1 [] = "ABCDEFGG";
  char s2 [] = "XYZ";
  cout << "Before strcat (s1, s2): \n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen (s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen (s2) << endl;
  strcat (s1, s2);
  cout << "After strcat (s1, s2): \n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen (s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen (s2) << endl;
}
```

```
Before strcat (s1, s2):
  s1 = [ABCDEFGG], length=7
```

```

s2 = [XYZ], length=3
After strcat (s1, s2):
s1 = [ABCDEFGXYZ], length=10
s2 = [XYZ], length=3

```

调用 `strcat (s1, s2)` 将 XYZ 追加到 s1 的末尾, 如图 8.11 所示。由于 s2 的长度为 3, `strcat (s1, s2)` 复制 4 字节 (包括空字符, 表示为  $\emptyset$ ), 覆盖了 s1 的空字符和其后的 3 字节, 使 s1 的长度变为 10。

如果复制 s2 所需的在 s1 后面的字节正在被其他对象使用, 则 s1 中的所有字节及追加的 s2 都会被复制到某个未分配的内存区域中。

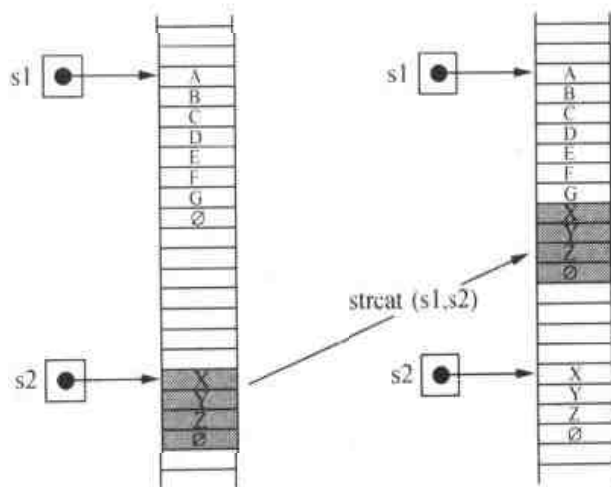


图 8.11 strcat 示例

### 例 8.18 第二个字符串连接函数 `strncat ()`

本例跟踪函数 `strncat (s1, s2, n)` 的调用过程:

```

#include <cstring>
#include <iostream>
using namespace std;
int main ()
{ // strncat () 函数的测试驱动程序
  char s1 [] = "ABCDEFGG";
  char s2 [] = "XYZ";
  cout << "Before strncat (s1, s2, 2): \n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen (s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen (s2) << endl;
  strncat (s1, s2, 2);
  cout << "After strncat (s1, s2, 2): \n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen (s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen (s2) << endl;
}

```

```

Before strncat (s1, s2, 2):
s1 = [ABCDEFGG], length=7
s2 = [XYZ], length=3
After strncat (s1, s2, 2):
s1 = [ABCDEFGXY], length=9
s2 = [XYZ], length=3

```

调用 `strncat (s1, s2, 2)` 将 XY 追加到 s1 的末尾, 如图 8.12 所示。由于 s2 的长度为 3, `strncat (s1, s2, 2)` 复制 2 字节覆盖了 s1 的空字符和其后的字节, 然后将空字符放在下一个字节以结束字符串 s1, 使 s1 的长度变为 9。(如果 s1 后面的这两个字节正在被其他对

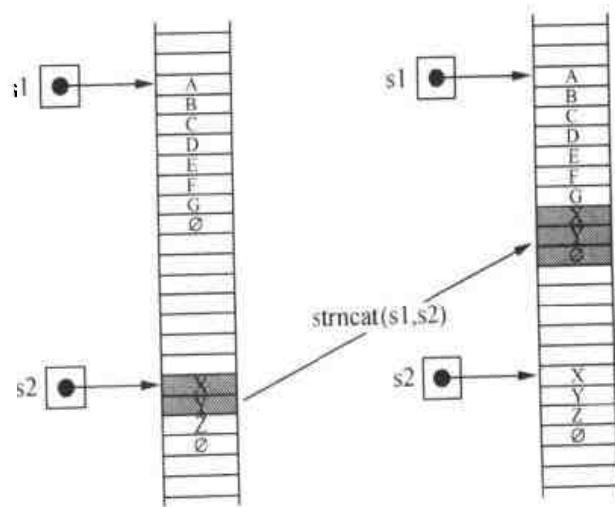


图 8.12 strncat 示例



象使用, 则整个的 10 个字符 ABCDEFGXY $\phi$  都会被复制到某个未分配的内存区域中。)

下一个例子说明字符串标记函数, 它的作用是在一个给定的字符串中指明“标记”: 例如, 一个句子中的单词。

### 例 8.19 字符串标记函数 strtok ()

本例说明如何使用 strtok () 从一个句子中抽取单个的单词。

```
#include <cstring>
#include <iostream>
using namespace std;
int main ()
{ // strtok () 函数的测试驱动程序
  char s [] = "Today's date is March 12, 2000.";
  char* p;
  cout << "The string is: [" << s << "]" << "\nIts tokens are: \n";
  p = strtok (s, " ");
  while (p)
  { cout << "  [" << p << "]" << "\n";
    p = strtok (NULL, " ");
  }
  cout << "Now the string is: [" << s << "]" << "\n";
}
```

```
The string is: [Today's date is March 12, 2000.]
Its tokens are:
  [Today's]
  [date]
  [is]
  [March]
  [12,]
  [2000.]
Now the string is: [Today's]
```

如图 8.13 所示, 调用  $p = \text{strtok}(s, " ")$  设置指针  $p$  指向字符串  $s$  中的第一个标记, 并将跟在 "Today's" 后面的空格符变为空字符 ' $\backslash 0$ ' (在图 8.13 中表示为  $\phi$ ), 这将同时使  $s$  和  $p$  指向字符串 "Today's"。然后每次成功调用  $p = \text{strtok}(NULL, " ")$  都使指针  $p$  前进到跟在新的空字符后的下一个非空格的字符处, 将它所经过的每个空格符修改为空字符, 并将跟在  $*p$  后面的第一个空格符修改为空字符, 这将使  $p$  指向下一个原来由空格符定界而现在由空字符定界的子串, 重复上面的操作, 直到  $p$  指向原来的字符串  $s$  末尾的空字符, 这使  $p$  为空 (即 0), 将终止 while 循环。所有的调用 strtok () 将影响原来的字符串  $s$ , 将其中的每个空格符修改为空字符, 这样就“标记”了字符串  $s$ , 将它变为一系列单独的标记字符串, 而  $s$  只指向其中的第一个串。

注意函数 strtok () 改变了它标记的字符串, 因此, 如果想在标记一个字符串之后使用原来的字符串, 则应该使用 strcpy () 复制它。

还要注意 strtok () 函数的第二个参数是一个字符串, 函数使用这个字符串中的所有字符作为第一个字符串的定界符。例如, 为了标记  $s$  中的单词, 可以使用  $\text{strtok}(s, ".,:; .")$ 。

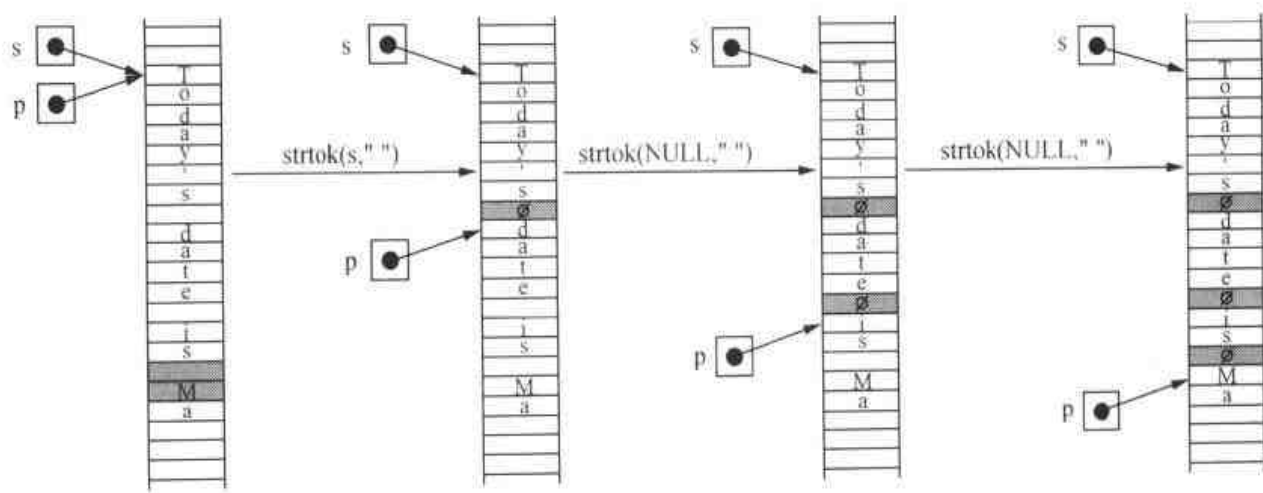


图 8.13 strtok 函数工作过程

strpbrk() 函数将一个字符串中的字符作为一个字符集合使用。它推广了 strchr() 函数，在第一个字符串中寻找第二个字符串中的任意字符首次出现的位置。

### 例 8.20 strpbrk() 函数

```
#include <cstring>
#include <iostream>
using namespace std;
int main ()
{ char s [] = "The Mississippi is a long river.";
  cout << "s = \"\" << s << "\"\n";
  char * p = strpbrk (s, "nopqr");
  cout << "strpbrk (s, \"nopqr\") points to s [" << p - s << "] .\n";
  p = strpbrk (s, "NOPQR");
  if (p == NULL) cout << "strpbrk (s, \"NOPQR\") returns NULL.\n";
}
```

```
s = "The Mississippi is a long river."
strpbrk (s, "nopqr") points to s [12]
strpbrk (s, "NOPQR") returns NULL
```

调用 strpbrk(s, "nopqr") 返回五个字符 'n', 'o', 'p', 'q', 'r' 中的任意一个在 s 中首次出现的位置，其中找到的第一个是在 s[12] 的 'p'。

调用 strpbrk(s, "NOPQR") 返回空指针，因为这五个字符都没有在 s 中出现。

下表总结了在 <cstring> 中定义的非常有用的函数，注意 size\_t 是一个在 <cstring> 中定义的特殊整数。

memcpy()	void * memcpy (void * s1, const void * s2, size_t n); 用 *s2 中的前 n 个字节替换 *s1 中的前 n 个字节，返回 s1。
strcat()	char * strcat (char * s1, const char * s2); 将 s2 追加到 s1 之后，返回 s1。
strchr()	char * strchr (const char * s, int c); 返回一个指针，指向 c 在 s 中出现的第一个位置。如果未找到 c，则返回空指针。
strcmp()	int strcmp (const char * s1, const char * s2); 将 s2 与 s1 相比，按照词典顺序，根据 s1 是小于、等于或大于 s2，返回一个负整数、0 或正整数。

(续表)

strcpy ()	char* strcpy (char* s1, const char* s2); 用 s2 代替 s1, 返回 s1;
strspn ()	size_t strspn (char* s1, const char* s2); 返回 s1 中从 s[0] 开始, 并且不包含 s2 中的任意字符的最长子串的长度。
strlen ()	size_t strlen (const char* s); 返回 s 的长度, 它是从 s[0] 开始, 到第一个空字符之间的所有字符的个数。
strncat ()	char* strncat (char* s1, const char* s2, size_t n); 追加 s2 中的前 n 个字符到 s1, 返回 s1, 如果 $n \geq \text{strlen}(s2)$ , 则 strncat (s1, s2, n) 的作用与 strcat (s1, s2) 相同。
strncmp ()	int strncmp (const char* s1, const char* s2, size_t n); 将 s1 中的前 n 个字符与 s2 的前 n 个字符相比, 按照词典顺序, 根据第一个子串是小于、等于或大于第二个子串, 返回一个负整数、0 或正整数。如果 $n \geq \text{strlen}(s2)$ , 则 strncmp (s1, s2, n) 的作用与 strcmp (s1, s2) 相同。
strncpy ()	char* strncpy (char* s1, const char* s2, size_t n); 用 s2 中的前 n 个字符代替 s1 中的前 n 个字符, 返回 s1。如果 $n \leq \text{strlen}(s1)$ , 则不影响 s1 的长度; 如果 $n \geq \text{strlen}(s2)$ , 则 strncpy (s1, s2, n) 的作用与 strcpy (s1, s2) 相同。
strpbrk ()	char* strpbrk (const char* s1, const char* s2); 返回 s2 中的任意字符第一次出现在 s1 中的位置, 如果 s2 中的任意一个字符都未在 s1 中出现, 则返回空指针。
strchr ()	char* strchr (const char* s, int c); 返回一个指针, 指向 c 在 s 中最后一次出现的位置。如果 c 不在 s 中, 则返回空指针。
strspn ()	size_t strspn (char* s1, const char* s2); 返回 s1 中从 s1[0] 开始, 并且包含 s2 中的单独字符的最长子串的长度。
strstr ()	char* strstr (const char* s1, const char* s2); 返回 s2 作为 s1 的子串首次出现在 s1 中的位置。如果 s2 不在 s1 中, 则返回空指针。
strtok ()	char* strtok (char* s1, const char* s2); 用 s2 中的字符作为定界符标记 s1, 在首次调用 strtok (s1, s2) 之后, 每次成功调用 strtok (NULL, s2) 都返回一个指针, 指向在 s1 中找到的下一个标记。这些调用会修改 s1, 用空字符 '\0' 代替每个定界符。

## 复 习 题

### 8.1 考虑以下对 s 的声明:

```

char s[6];
char s[6] = {'H', 'e', 'l', 'l', 'o'};
char s[6] = "Hello";
char s[];
char s[] = new char[6];
char s[] = {'H', 'e', 'l', 'l', 'o'};
char s[] = "Hello";
char s[] = new ("Hello");
char* s;
char* s = new char[6];
char* s = {'H', 'e', 'l', 'l', 'o'};
char* s = "Hello";

```

```
char * s = new ("Hello");
```

- a. 其中哪些正确地声明了一个 C++ 字符串?
- b. 其中哪些正确地声明了一个 C++ 字符串, 要求串长为 5, 初始化为字符串 "Hello", 并且在编译时分配内存?
- c. 其中哪些正确地声明了一个 C++ 字符串, 要求串长为 5, 初始化为字符串 "Hello", 并且在运行时分配内存?
- d. 其中哪些正确地声明了一个 C++ 字符串, 可以作为一个函数的参数?

## 8.2 使用语句

```
cin >> s;
```

读入输入的 "Hello, World!" 到字符串 s 有什么错误?

## 8.3 以下代码的打印结果是什么:

```
char s[] = "123 W. 42nd St., NY, NY 10020-1095";  
int count = 0;  
for (char* p = s; *p; p++)  
    if (isupper (*p)) ++count;  
cout << count << endl;
```

## 8.4 以下代码的打印结果是什么:

```
char s[] = "123 W. 42nd St., NY, NY 10020-1095";  
for (char* p = s; *p; p++)  
    if (isupper (*p)) *p = tolower (*p);  
cout << s << endl;
```

## 8.5 以下代码的打印结果是什么:

```
char s[] = "123 W. 42nd St., NY, NY 10020-1095";  
for (char* p = s; *p; p++)  
    if (isupper(*p)) (*p)++;  
cout << s << endl;
```

## 8.6 以下代码的打印结果是什么:

```
char s[] = "123 W. 42nd St., NY, NY 10020-1095";  
int count = 0;  
for (char* p = s; *p; p++)  
    if (ispunct(*p)) ++count;  
cout << count << endl;
```

## 8.7 以下代码的打印结果是什么:

```
char s[] = "123 W. 42nd St., NY, NY 10020-1095";  
for (char* p = s; *p; p++)  
    if (ispunct(*p)) *(p-1) = tolower(*p);  
cout << s << endl;
```

## 8.8 如果 s1 和 s2 的类型都为 char\*, 以下两个语句的区别是什么?

```
char* ;
    s1 = s2;
strcpy (s1, s2);
```

8.9 如果 first 中包含字符串 “Rutherford”，且 last 中包含字符串 “Hayes”，则以下调用的结果是什么：

- a. int n = strlen (first);
- b. char\* s1 = strchr (first, 'r');
- c. char\* s1 = strrchr (first, 'r');
- d. char\* s1 = strpbrk (first, "rstuv");
- e. strcpy (first, last);
- f. strcpy (first, last, 3);
- g. strcat (first, last);
- h. strcat (first, last, 3);

8.10 以下语句将什么值赋给 n：

- a. int n = strspn ("abecedarian", "abcde");
- b. int n = strspn ("beefeater", "abcdef");
- c. int n = strspn ("baccalaureate", "abc");
- d. int n = strspn ("baccalaureate", "rstuv");

8.11 以下代码的打印结果是什么：

```
char* s1 = "ABCDE";
char* s2 = "ABC";
if (strcmp(s1,s2) < 0) cout << s1 << " < " << s2 << endl;
else cout << s1 << " >= " << s2 << endl;
```

8.12 以下代码的打印结果是什么：

```
char* s1 = "ABCDE";
char* s2 = "ABCE";
if (strcmp(s1,s2) < 0) cout << s1 << " < " << s2 << endl;
else cout << s1 << " >= " << s2 << endl;
```

8.13 以下代码的打印结果是什么：

```
char* s1 = "ABCDE";
char* s2 = "";
if (strcmp(s1,s2) < 0) cout << s1 << " < " << s2 << endl;
else cout << s1 << " >= " << s2 << endl;
```

8.14 以下代码的打印结果是什么：

```
char* s1 = " ";
char* s2 = " ";
if (strcmp(s1,s2) == 0) cout << s1 << " == " << s2 << endl;
else cout << s1 << " != " << s2 << endl;
```

## 习 题

8.1 说明为什么下面改写例 8.12 的程序不能运行?

```
int main()
{ char name[10][20], buffer[20];
  int count = 0;
  while (cin.getline(buffer,20))
    name[count++] = buffer;
  --count;
  cout << "The names are:\n";
  for (int i = 0; i < count; i++)
    cout << "\t" << i << ". [" << name[i] << "]" << endl;
}
```

8.2 写出 strcpy () 函数

8.3 写出 strcat () 函数

8.4 写出并测试一个函数, 返回传递给它的一个英语单词的复数形式。

8.5 写出一个程序, 读入一个名字的序列, 每行一个名字, 然后将它们排序并打印结果。

8.6 写出并测试一个函数, 不复制任意字符而将一个字符串在原地倒序存放

8.7 使用

```
while (cin >> word)
```

代替

```
do..while (· word)
```

修改例 8.3 中的程序, 并运行修改后的程序。

8.8 写出 strchr () 函数。

8.9 写出一个函数, 返回一个给定字符在一个给定字符串中出现的次数。

8.10 写出并测试 strrchr () 函数。

8.11 写出并测试 strstr () 函数。

8.12 写出并测试 strncpy () 函数。

8.13 写出并测试 strcat () 函数。

8.14 写出并测试 strcmp () 函数。

8.15 写出并测试 strncmp () 函数。

8.16 写出并测试 strspn () 函数。

8.17 写出并测试 strcspn () 函数。

8.18 写出并测试 strpbrk () 函数。

8.19 写出一个函数, 返回在一个给定的字符串中包含一个给定字符的单词的个数 (参见例 8.19)。

8.20 首先, 预测以下程序将对字符串 s 作什么操作 (参见例 8.19), 然后运行该程序验证

你的预测。

```
int main()
{ char s[] = "###ABCD#EFG##HIJK#L#MN####O#P#####";
  char* p;
  cout << "The string is: [" << s << "]\nIts tokens are:\n";
  p = strtok(s, "#");
  while (p)
  { cout << "\t[" << p << "]\n";
    p = strtok(NULL, "#");
  }
  cout << "Now the string is: [" << s << "]\n";
}
```

- 8.21 写出一个程序，读入一行文字，然后将其中所有的字母大写并打印结果。
- 8.22 写出一个程序，读入一行文字，然后将其中所有的空格符删除并打印结果。
- 8.23 写出一个程序，读入一行文字，然后打印出其中的单词个数。
- 8.24 写出一个程序，读入一行文字，然后颠倒每个单词的顺序并打印结果。例如，输入：

today is Tuesday

则输出为：

Tuesday is today

## 复习题答案

- 8.1 在 13 个声明中：

a. 以下是正确的 C++ 字符串声明：

```
char s[6];
char s[6] = {'H', 'e', 'l', 'l', 'o'};
char s[6] = "Hello";
char s[] = {'H', 'e', 'l', 'l', 'o'};
char s[] = "Hello";
char* s;
char* s = new char[6];
char* s = "Hello";
```

警告：最后一个声明仅仅定义了 s 为一个指向字符串常量的指针。

- b. 以下声明正确地声明了一个 C++ 字符串，要求串长为 5，初始化为字符串 "Hello"，并且在编译时分配内存：

```
char s[6] = {'H', 'e', 'l', 'l', 'o'};
char s[6] = "Hello";
char s[] = {'H', 'e', 'l', 'l', 'o'};
char s[] = "Hello";
char* s = "Hello"; //把 s 定义为一个指向一个字符串常量的指针
```

- c. 不可能在运行时像这样初始化一个 C++ 字符串。

- d. 以下声明正确地声明了一个 C++ 字符串，它可以作为一个函数的参数：

```
char s[];
char * s;
```

8.2 该语句将只读到第一个空白符。对于给定的输入，它将"Hello,"赋值给 s。

8.3 该代码将计算字符串 s 中大写字母的个数，所以打印结果是 6。

8.4 该代码将字符串 s 中的所有大写字母变为小写字母，所以打印结果为：

123 W. 42nd st., ny, ny 10020-1095

注意为了修改 \* p，必须将它赋值为函数的返回值：

```
*p = tolower (p);
```

8.5 该代码将所有的大写字母加 1，将 W 变为 X，将 S 变为 T，等等：

123 X. 42nd St., OZ, OZ 10020-1095

8.6 该代码计算在字符串 s 中的所有标点符号的个数，所以打印结果是 5。

8.7 该代码将所有后跟标点符号的字符改写为所跟的标点符号，

123 .. 42nd S., , N., NY 1002- -1095

8.8 赋值语句 s1 = s2 仅使 s1 成为 s2 的同义字，即它们指向同一个字符；调用 strcpy (s1, s2) 真正地将 s2 中的字符复制到 s1 中，即复制字符串。

8.9 a. 将整数 10 赋给 n。

b. 将子串"rford"赋给 s1。

c. 将子串"rd"赋给 s1。

d. 将子串"utherford"赋给 s1。

e. 将 last 复制到 first，所以 first 也将是字符串"Hayes"。

f. 将子串"Hay"复制到 first 的前面，使 first 变为"Hayherford"。

g. 将 last 追加到 first 末尾，使 first 变为"RutherfordHayes"。

h. 将子串"Hay"追加到 first 末尾，使 first 变为"RutherfordHay"。

8.10 a. 7

b. 6

c. 5

d. 7

8.11 打印结果为：ABCDE > = ABC

8.12 打印结果为：ABCDE < ABCE

8.13 打印结果为：ABCDE > =

8.14 打印结果为：! =

## 习题答案

8.1 它不能运行是因为赋值语句



```
name[count] = buffer;
```

将相同的指针赋给每个字符串 name[0], name[1] 等, 数组不能这样赋值。要将一个数组复制到另一个数组中, 需使用 strcpy() 或 strncpy()。

## 8.2 该函数将字符串 s2 复制到 s1 中:

```
char* strcpy(char* s1, const char* s2)
{ char* p; for (p=s1; *s2; )
    *p++ = *s2++;
    *p = '\0';
    return s1;
}
```

指针 p 初始化后指向 s1 的开头。在每次 for 循环中, 字符 \*s2 被复制到字符 \*p, 然后 s2 和 p 都加 1, 继续循环直到 \*s2 为 0 (即空字符 '\0')。然后, 通过将空指针赋给 \*p 使它追加到 s1 的末尾 (当循环终止时, 指针 p 指向复制的最后一个字符后面的字节)。注意, 该函数不能分配新的存储空间, 所以它的第一个参数 s1 应该定义为与 s2 长度相同。

## 8.3 该函数将 s2 前 n 个字符追加到 s1 的末尾, 除了用第三个参数限制复制字符的个数, 它与函数 strcat() 基本相同:

```
char* strncat(char* s1, const char* s2, size_t n)
{ char* end; for (end=s1; *end; end++) // 指到 s1 的结尾
    ;
    char* p; for (p=s2; *p && p-s2<n; )
        *end++ = *p++;
    *end = '\0';
    return s1;
}
```

第一个 for 循环找到 s1 的末尾, 这是开始追加 s2 的位置。第二个 for 循环将字符从 s2 中复制到 s1 的末尾, 注意附加条件  $p - s2 < n$  是如何将复制的字符个数限制到 n 的。表达式  $p - s2$  等于被复制的字符个数, 因为它是 p (指向下一个被复制的字符) 和 s2 (指向字符串 s2 的开始) 的差。注意, 该函数不能分配新的存储空间, 它要求 s1 至少有 k 个多余的字节空间, 其中 k 是 n 和字符串 s2 长度的最小值。

## 8.4 需要测试单词的最后一个字母和倒数第二个字母是否变为复数形式。使用指针 p 和 q 来访问这些字母。

```
void pluralize(char* s)
{ int len = strlen(s);
  char* p = s + len - 1; // 最后一个字母
  char* q = s + len - 2; // 最后两个字母
  if (*p == 'h' && (*q == 'c' || *q == 's')) strcat(p, "es");
  else if (*p == 's') strcat(p, "es");
  else if (*p == 'y')
      if (isvowel(*q)) strcat(p, "s");
      else strcpy(p, "ies");
  else if (*p == 'z')
```

```

        if (isvowel(*q)) strcat(p, "zes");
        else strcat(p, "es");
    else strcat(p, "s");
}

```

第二个测试依赖于倒数第二个字母是否为元音字母，所以定义一个小的布尔型函数 `isvowel()` 来测试这个条件。

```

bool isvowel(char c)
{ return (c=='a' || c=='e' || c=='i' || c=='o' || c=='u');
}

```

测试程序反复地读入一个单词，打印它，将其变为复数，再打印。当用户输入一个空格符时，循环终止。

```

bool pluralize(char*);
int main()
{ char word[80];
  for (;;)
  { cin.getline(word, 80);
    if (*word == ' ') break;
    cout << "\tThe singular is [" << word << "].\n";
    pluralize(word);
    cout << "\t The plural is [" << word << "].\n";
  }
}

```

```

wish
    The singular is [wish] .
    The plural is [wishes] .
hookah
    The singular is [hookah] .
    The plural is [hookahs] .
bus
    The singular is [bus] .
    The plural is [buses] .
toy
    The singular is [toy] .
    The plural is [toys] .
navy
    The singular is [navy] .
    The plural is [navies] .
quiz
    The singular is [quiz] .
    The plural is [quizzes] .
quartz
    The singular is [quartz] .
    The plural is [quartzes] .
computer
    The singular is [computer] .
    The plural is [computers] .

```

- 8.5 假设一个名字不超过 20 个字符，并且不超过 25 个名字。一次读入所有的输入并将其存为一个 buffer。由于所有的名字都是以一个空字符结尾的，这个 buffer 需要足够的空

间来存放  $25 * (20 + 1) + 1$  个字符 (21 是 20 个字符加上最后一个空字符)。程序被模块化为 5 个函数调用。调用 `input (buffer)` 将所有输入读入 `buffer`。调用 `tokenize (name, numNames, buffer)` “标记” 这个 `buffer`，存放指向名字的指针到数组 `name` 中，并且返回在 `numNames` 中的名字的个数。调用 `print (name, numNames)` 打印存放在 `buffer` 中的所有名字。调用 `sort (name, numNames)` 通过存放在数组 `name` 中的指针，对存放在 `buffer` 中的名字进行间接排序。

```
#include <cstring>
#include <iostream>
using namespace std;
const int NAME_LENGTH = 20;
const int MAX_NUM_NAMES = 25;
const int BUFFER_LENGTH = MAX_NUM_NAMES*(NAME_LENGTH + 1);
void input(char* buffer);
void tokenize(char** name, int& numNames, char* buffer);
void print(char** name, int numNames);
void sort(char** name, int numNames);
int main()
{ char* name[MAX_NUM_NAMES];
  char buffer[BUFFER_LENGTH+1];
  int numNames;
  input(buffer);
  tokenize(name, numNames, buffer);
  print(name, numNames);
  sort(name, numNames);
  print(name, numNames);
}
```

整个输入是通过调用 `cin.getline (buffer, BUFFER_LENGTH, '$')` 实现的，它连续读入字符，直到读入 '\$' 符时结束，将所有的字符存入 `buffer`。

```
void input(char* buffer)
{ // 把 25 个字符串读入到 buffer 中:
  cout << "Enter up to " << MAX_NUM_NAMES << " names, one per"
    << " line. Terminate with '$'.\nNames are limited to "
    << NAME_LENGTH << " characters.\n";
  cin.getline(buffer, BUFFER_LENGTH, '$');
}
```

`tokenize ()` 函数使用 `strtok ()` 函数扫描 `buffer`，“标记”以换行符 '\n' 结尾的每个子串，并将其地址存放在 `name` 数组中。for 循环反复执行，直到 `p` 指向标记 '\$'。注意函数的 `name` 参数被声明为 `char * *` 型，因为它是一个指向 `char` 型的指针数组。还要注意计数器 `n` 被声明为 `int&` (通过引用传递)，以使它的新值被返回 `main ()`。

```
void tokenize(char** name, int& n, char* buffer)
{ // 把 buffer 中每个字符串的地址复制到 name 数组中:
  char* p = strtok(buffer, "\n"); // p 指向每个符号
  for (n = 0; p && *p != '$'; n++)
  { name[n] = p;
    p = strtok(NULL, "\n");
  }
}
```

print () 和 sort () 函数与前面看到的类似,除了这里的操作是间接的。两个函数都是在数组 name 上操作的。

```
void print(char** name, int n)
{ // 打印 buffer 中存储的 n 个名字:
  cout << "The names are:\n";
  for (int i = 0; i < n; i++)
    cout << "\t" << i+1 << ". " << name[i] << endl;
}

void sort(char** name, int n)
{ // 将buffer中存储的 n 个名字排序:
  char* temp;
  for (int i = 1; i < n; i++) //冒泡排序
    for (int j = 0; j < n-i; j++)
      if (strcmp(name[j], name[j+1]) > 0)
      { temp = name[j];
        name[j] = name[j+1];
        name[j+1] = temp;
      }
}
```

Enter up to 25 names, one per line. Terminate with '\$'.  
Names are limited to 20 characters.

Washington, George  
Adams, John  
Jefferson, Thomas  
Madison, James  
Monroe, James  
Adams, John Quincy  
Jackson, Andrew

\$The names are:

1. Washington, George
2. Adams, John
3. Jefferson, Thomas
4. Madison, James
5. Monroe, James
6. Adams, John Quincy
7. Jackson, Andrew

The names are:

1. Adams, John
2. Adams, John Quincy
3. Jackson, Andrew
4. Jefferson, Thomas
5. Madison, James
6. Monroe, James
7. Washington, George

在试运行中,用户输入 7 个名字和标记 '\$',然后这些名字被打印,存储,并再次打印。

- 8.6 该函数首先定位在字符串末尾,然后交换第一个字符和最后一个字符,再交换第二个字符和倒数第二个字符,依此类推。

```

void reverse(char* s)
{ char* end, temp;
  for (end = s; *end; end++)
    ; //找到s的末尾
  while (s < end - 1)
  { temp = *--end;
    *end = *s;
    *s++ = temp;
  }
}

```

测试程序使用函数 `getline()` 读入字符串，然后打印出来，再倒序存放该字符串，最后再次打印。

```

void reverse(char*);
int main()
{ char string[80];
  cin.getline(string, 80);
  cout << "The string is [" << string << "].\n";
  reverse(string);
  cout << "The string is [" << string << "].\n";
}

```

```

Today is Wednesday.
The string is [Today is Wednesday.] .
The string is [.yadsendew si yadot] .

```

```

8.7 int main()
{ char word[80];
  while (cin >> word)
    if (*word) cout << "\t\"" << word << "\"\n";
}

```

```

Today is Wednesday.
    "Today"
    "is"
    "Wednesday."
~Z

```

```

8.8 char* Strchr(const char* s, int c)
{ for (const char* p=s; p && *p; p++)
  if (*p==c) return (char*)p;
  return 0;
}

```

```

8.9 int numchr(const char* s, int c)
{ int n=0;
  for (const char* p=s; p && *p; p++)
    if (*p==c) ++n;
  return n;
}

```

- ```

8.10 char* Strchr(const char* s, int c)
{ const char* pp=0;
  for (const char* p=s; p && *p; p++)
    if (*p==c) pp = p;
  return (char*)pp;
}

8.11 char* Strstr(const char* s1, const char* s2)
{ if (*s2==0) return (char*)s1; // s2 是空的字符串
  for ( ; *s1; s1++)
    if (*s1==*s2)
      for (const char* p1=s1, * p2=s2; *p1==*p2; p1++, p2++)
        if (*(p2+1)==0) return (char*)s1;
  return 0;
}

8.12 char* Strncpy(char* s1, const char* s2, size_t n)
{ char* p=s1;
  for ( ; n>0 && *s2; n--)
    *p++ = *s2++;
  for ( ; n>0; n--)
    *p++ = 0;
  return s1;
}

8.13 char* Strcat(char* s1, const char* s2)
{ char* p=s1;
  for ( ; *p; p++)
    ;
  for ( ; *s2; p++, s2++)
    *p = *s2;
  *p = 0;
  return s1;
}

8.14 int Strcmp(char* s1, const char* s2)
{ for ( ; *s1==*s2; s1++, s2++)
  if (*s1==0) return 0;
  return (int)(*s1-*s2);
}

8.15 int Strncmp(char* s1, const char* s2, size_t n)
{ for ( ; n>0; s1++, s2++, n--)
  if (*s1!=*s2) return (int)(*s1-*s2);
  else if (*s1==0) return 0;
  return 0;
}

8.16 size_t Strspn(const char* s1, const char* s2)
{ const char *p1, *p2;
  for ( p1 = s1 ; *p1; p1++)
    for ( p2 = s2 ; ; p2++ )

```

- ```

        if ( *p2 == '\0' ) // 到达 s2 的结尾, 未发现匹配
            return ( p1 - s1 ) ; // 因此 *p1 不在 s2[] 中
        else if ( *p1 == *p2 ) // *p1 不是这一个
            break ; // 中止内循环
    return ( p1 - s1 ) ; // 返回 s1 的长度
}

```
- 8.17 `size_t Strcspn(const char* s1, const char* s2)`
- ```

{ const char *p1, *p2;
  for ( p1 = s1 ; *p1; p1++ )
    for ( p2 = s2 ; *p2 ; p2++ )
      if ( *p1 == *p2 ) // 在 s2[] 中找到 *p1
        return ( p1 - s1 ) ; // 并且 p1-s1 是它的下标
  return ( p1 - s1 ) ; // 返回 s1 的长度
}

```
- 8.18 `char* Strpbrk(const char* s1, const char* s2)`
- ```

{ const char *p1, *p2;
  for ( p1 = s1 ; *p1; p1++ )
    for ( p2 = s2 ; *p2 ; p2++ )
      if ( *p1 == *p2 ) // 在 s2[] 中找到 *p1
        return (char*) p1 ; // 因此返回它的地址
  return NULL ; // 没有 s1 字符在 s2[] 中
}

```
- 8.19 `int freqInWords(const char* sentence, char ch)`
- ```

{ int count = 0 ;
  char* copy = new char[ strlen(sentence) ] ;
  copy = strcpy( copy, sentence ) ;
  if ( copy == NULL ) return 0 ;
  char *p = strtok(copy, "\t\n\v\f\r" ) ;
  while (p) {
    for ( int i = 0 ; p[i] ; i++ )
      if ( p[i] == ch ) // 在当前的单词中找到 ch
        { count++ ; // 由 p 引用
          break ; // 由当前的单词结束
        } // end if ( p[i] == ch )
    p = strtok(NULL, "\t\n\v\f\r" ) ; // 前进到下一个单词
  } // 结束 while(p) 循环
  return count ; //
}

```
- 8.20
- 8.21 `void capitalize(char* s)`
- ```

{ if (s == NULL) return;
  for (char* p=s; *p; p++)
    if (*p>='a' && *p<='z') *p = (char)( *p - 'a' + 'A');
}

```
- 8.22 `void removeBlanks( char* s)`
- ```

{ if ( s == NULL ) return ;
  int j = 0 ;
  for ( int i = 0; s[i] ; i++ )

```

```

        if ( s[i] != ' ' ) s[j++] = s[i] ;
        s[j] = '\0' ;
    }

```

```

8.23 int numWords( const char* s)
{ if ( s == NULL ) return 0 ;
  int wordCount = 0 ;
  char * Copy = new char[ strlen(s) ] ;
  Copy = strcpy( Copy, s ) ;
  char * p = strtok( Copy, "\n \v\t\f\r" ) ;
  while ( p )
  { char ch0 = p[0]; //检查第一个字符是否字母
    if ( ((ch0 >= 'a') && (ch0 <= 'z')) | //小写字母
          ((ch0 >= 'A') && (ch0 <= 'Z')) ) //大写字母
      wordCount++ ;
    p = strtok( NULL, "\n \v\t\f\r" ) ;
  }
  return wordCount ;
}

8.24 char* reverseWords(char* reverseS, const char* s)
{ if ( (reverseS == NULL) || (s == NULL) ) return NULL;
  char * Copy = new char[ strlen(s) ] ;
  Copy = strcpy( Copy, s ) ;
  char * currentReverse = new char[ strlen(s) ] ;
  char * revPtr = reverseS ;
  *revPtr = '\0' ; // reverse starts with no words
  char * pS;
  pS = strtok(Copy, " \t" ) ; //由 space 或 tab 分开的单词
  while ( pS )
  { // reverseS = currentWordInS + currentReverse
    currentReverse = strcpy( currentReverse, revPtr ) ;
    revPtr = addWords( revPtr, pS, currentReverse ) ;
    pS = strtok( NULL, " \t" ) ; //把 pS 推进到 s 中的下一个单词
  } //结束 while(pS) 循环
  return revPtr ;
}

char* addWords( char* leftPLUSright, const char* left,
               const char* right )
{ char * both = leftPLUSright ;
  const char * pLeft = left ;
  const char * pRight = right ;
  while ( *pLeft )
    *(both++) = *(pLeft++) ;
  if ( *left && *right ) //两个单词都非空
    *(both++) = ' ' ; //在中间放入一个空格
  while ( *pRight )
    *(both++) = *(pRight++) ;
  *both = '\0' ; //用空字符中止新的字符串
  return leftPLUSright ;
}

```



## 第9章 标准 C++ 字符串

### 9.1 引言

第8章中描述的经典的C字符串是C++中很重要的一部分。它们提供了一个十分有效的快速的数据处理方法。但作为一般的数组，C字符串的效率要付出代价。

标准C++字符串为C字符串提供了一个安全的替代。

### 9.2 格式化的输入

回忆C++中流的思想，它是作为数据传输的管道。输入通过了一个istream对象，输出通过了一个ostream对象。istream类定义了对对象的行为，如cin。最一般的行为是取出运算符>>（也叫做输入运算符）的应用。它有两个操作数：从其中取出字符的istream对象和它把从这些字符中形成的相应值复制到的对象。从初始输入字符形成一个类型值的过程叫做格式化。

#### 例9.1 取出运算符>> 执行格式化输入

假设代码

```
int n;  
cin >> n;
```

执行时输入46。这个输入实际上包含了七个字符：' ',' ',' ',' ','4','6','\n'（四个空格后面是4、6和换行符）如图9.1它可以看做正在通过输入流。流对象cin每次检查一个字符。如果它看到的第一个字符是空白字符（空格、标号、换行符等等），它取出空白字符，忽略它。它继续取出并忽略空白字符，直到遇到一个非空白字符。在这个例子中，第一个非空白字符是'4'。由于表达式cin >> n的第二个操作数是int类型，对象cin正在寻找能生成一个整数的数字。因此，在“吃掉”前面的任一个空白符之后，它期望发现12个字符'+','-'，

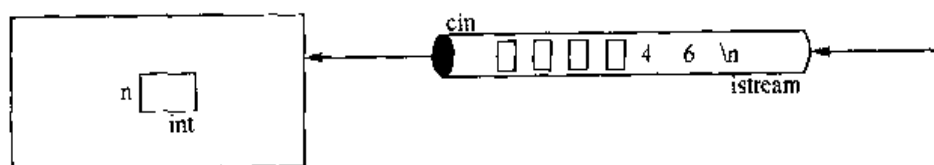


图9.1 取出运算符<<工作

'0','1','2','3','4','5','6','7','8','9'中的一个。如果它遇到了其他 244 个字符中的任一个,就失败了。在这个例子中它找到了'4'。因此,它取出它,然后继续,期望找到更多的数字。只要它只遇到了数字,它就继续取出它们。当它看到非数字时就停于此,把那个非数字留在流中。在这个例子中,这就意味着 cin 将正好取出六个字符:四个空格,'4','6'。它把四个空格去掉,然后把'4','6'组合成整数 46,然后,它把这个值复制给对象 n,如图 9.2 所示。

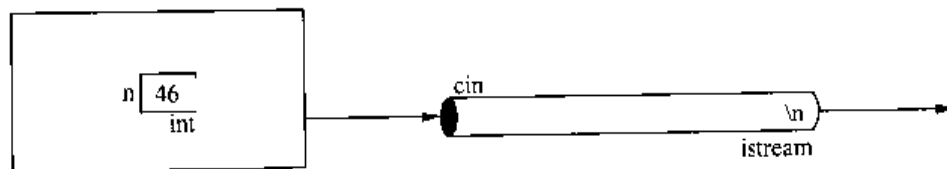


图 9.2 取出运算符 >> 工作

取出操作完成后,换行符仍在输入流中。如果下次输入语句是另一个格式化输入,那么像其他的空白符一样换行符将被忽略。

取出运算符 >> 格式化了它通过输入流收到的数据。这意味着它从流中取出字符,并用它们来形成一个相同类型的值,作为它的第二个操作数。在这个过程中,它忽略了正在使用的字符前面的所有空白字符。这个规则的一个直接结果就是,用取出运算符来读空白符是不可能的。为了这一点必须作用一个非格式化输入函数。

根据输入是否成功,运算符表达式 `cin >> x` 在某种情况下能认为是布尔型的值,也就是 true 或者 false。这允许一个这样的表达式用来控制一个循环。

### 例 9.2 用取出运算符来控制一个循环

```
int main ()
{ int n;
  while (cin >> n)
    cout << "n = " << n << endl;
```

```
;
```

```
46
```

```
n = 46
```

```
22 44 66 88
```

```
n = 22
```

```
n = 44
```

```
n = 66
```

```
n = 88
```

```
33, 55, 77, 99
```

```
n = 33
```

只要整型数只是由空白符分开,循环就继续重复。第一个非空白符号逗号','导致了输入失败,因此也结束了循环。

## 9.3 非格式化输入

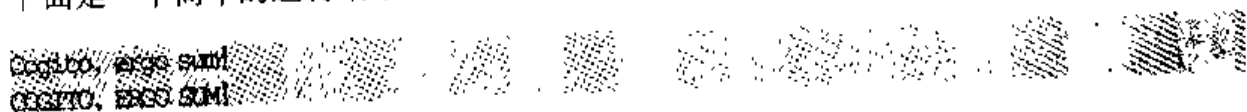
<iostream> 文件定义了几个输入字符和 C 字符串的函数，它们不跳过空白符。最普通的是用来读单个字符的 `cin.get()` 函数和用来读 C 字符串的 `cin.getline()` 函数。

### 例 9.3 用 `cin.get()` 函数输入字符

```
while (cin.get (c))
    { if (c >= 'a' && c <= 'z') c += 'A' - 'a'; // 大写字母 C
      cout.put (c);
      if (c == '\n') break;
    }
```

这个循环由输入表达式 (`cin.get (c)`) 控制。当输入流对象 `cin` 发现到了文件末尾 (通过 `Ctrl+Z` 或 `Ctrl+D` 交互地发信息)，表达式对 `false` 求值并停止了循环。在读取和处理完换行符 '`\n`' 后，循环也可以用一个 `break` 语句结束。if 语句只是把所有的小写字母大写化，`cout.put (c)` 语句输出字符。

下面是一个简单的运行结果：



### 例 9.4 用 `cin.getline()` 函数来输入 C 字符串

这个程序给出如何一行接一行地读文本信息到 C 字符串的一个数组中：

```
const int LEN = 32; // 最大字长
const int SIZE = 10; // 数组大小
typedef char Name [LEN]; // 定义 Name 为一个 C 字符串类型
int main ()
{ Name king [SIZE]; // 定义 king 为一个 10 个名字的数组
  int n = 0;
  while (cin.getline (king [n++], LEN) && n < SIZE)
  {
    -- n; // 现在 n 等于读入的名字数目
    for (int i = 0; i < n; i++)
      cout << '\t' << i << " " << king [i] << endl;
  }
```

对象 `king` 是一个类型为 `Name` 的 10 个对象的数组。typedef 把 `Name` 定义为一个 32 个字符的 C++ 字符串 (31 个非空) 的同义词。函数调用 `cin.getline (king [n++], LEN)` 从 `cin` 读取字符，直到它或者已取出了 `LEN - 1` 个字符，或者它遇到了第一次出现的换行符。它把这些字符复制到 C 字符串 `king [n]` 中。如果它遇到了换行符，它就取出并忽略它 (也就是说，它并不复制它到 C 字符串中)，然后 `n` 加 1，如图 9.3 所示。

注意 while 循环体是空的。循环 cin 检测到文件末尾或者 `n == SIZE` 时停止。由于 `n` 从 0 开始, 并且在最后一个名字读取后加 1, 所以它的值总是比读取的名字的数目大 1。因此它每次在最后减 1, 这样它的值才和读取的名字数目相等。

当输入是从下面的文本文件读取时, 输出是:

Kings.dat

```
Kenneth II (971 - 995)
Constantine III (995 - 997)
Kenneth III (997 - 1005)
Malcolm II (1005 - 1034)
Duncan I (1034 - 1040)
Macbeth (1040 - 1057)
Lulach (1057 - 1058)
Malcolm III (1058 - 1093)
```

图 9.3 Kings.dat 文件内容

```
1. Kenneth II (971 - 995)
2. Constantine III (995 - 997)
3. Kenneth III (997 - 1005)
4. Malcolm II (1005 - 1034)
5. Duncan I (1034 - 1040)
6. Macbeth (1040 - 1057)
7. Lulach (1057 - 1058)
8. Malcolm III (1058 - 1093)
```

## 9.4 标准 C++ 字符串类型

标准 C++ 在头文件中定义它的 `string` 类型。类型 `string` 的对象可以用几种方式声明和初始化:

```
string s1;                //s1 包含 0 个字符
string s2 = "New York";   //s2 包含 8 个字符
string s3 (60, '*');      //s3 包含 60 个星号
string s4 = s3;           //s4 包含 60 个星号
string s5 (s2, 4, 2);     //s5 是串 "Yo" 的第二个字符
```

如果串没有初始化, 就像这里的 `s1`, 那么它表示包含 0 个字符的空串。一个串可以如 C 字符串一样的方式初始化, 像这里的 `s2`。或者一个串可以初始化为拥有一个给定数目的相同的字符, 像这里的 `s3` 有 60 个星号。不像一个 C 字符串, C++ 字符串对象可以如这里的 `s4` 那样用另一个已有串对象的复制来初始化, 或者如 `s5` 用一个已有串的子串来初始化。注意标准的子串指定符有三部分: 父串 (这里的 `s2`), 开始字符 (这里的 `s2 [4]`) 和子串的长度 (这里的 2)。

格式化的输入对 C++ 字符串和对 C 字符串的作用一样: 跳过前面的空白部分, 输入在第一个以空白部分结束的单词的末尾结束。C++ 字符串有一个 `getline()` 函数和 C 字符串中的 `cin.getline()` 函数的作用一样。

```
string s = "ABCDEFGH";
getline (cin, s);    //把全行字符读入到 s 中
```

它们也和 C 字符串一样使用下标运算符:

```
char c = s [2];      //把 'C' 赋给 c
s [4] = '*';         //把 s 改变为 "ABCD*FGH"
```

注意数组下标总是算出有多少个字符在这个下标的字符前面。C++字符串可以转换为C字符串，如下：

```
const char* cs = s.c_str();    //把s转换成C字符串cs
c_str() 函数返回类型 const char*。
```

C++ string 类也定义了一个 length() 函数，它可以用来决定在一个串中存储有多少个字符，如下：

```
Cout << s.length() << endl;    //对串 s == "ABCD*FG"输出 7
C++ 字符串也可以像基础类型那样用关系运算符作比较，例如：
```

```
if (s2 < s5) cout << "s2 lexicographically precedes s5 \n";
while (s4 == s3)    // ...
```

也可以用操作符 + 和 += 来连接和增加串，例如：

```
string s6 = s + "HIJK";        //把 s6 变成了"ABCD*FGHIJK"
s2 += s5;                      //把 s2 变成了"New YorkYo"
```

substring() 函数可如下使用：

```
s4 = s6.substring(5, 3);        //把 s4 变成了"FGH"
```

erase() 和 replace() 函数的作用如下：

```
s6.erase(4, 2);                //把 s6 变成了"ABCDGHIJK"
s6.replace(5, 2, "xyz");        //把 s6 变成了"ABCDGxyzJK"
```

find() 函数返回一个给定子串第一次出现字符的下标，如：

```
string s7 = "Mississippi River basin";
cout << s7.find("si") << endl;    //输出 3
cout << s7.find("so") << endl;    //输出串的长度 23
```

如果 find() 函数失败，它返回它正在搜索的字符串的长度。

### 例 9.5 使用标准 C++ string 类型

这个代码在一个元音前的每个“t”后加入一个音节。例如，句子：

The first step is to study the status of the C++ Standard.

被句子：

The first step is to study the status of the C++ Standard.

替换。它使用了个名叫 is\_vowel() 的辅助布尔型函数。

```
string word;
int k;
while (cin >> word)
{ k = word.find("t") + 1;
  if (k < word.length() && is_vowel(word[k]))
    word.replace(k, 0, "eg");
```

```
cout << word << ' ';
}
```

while 循环由输入控制，当检测到文件末尾时结束。它每次读一个单词。如果找到了字母 t 并且它的后面跟着一个元音，那么在 t 和这个元音之间插入 e.g.。

## 9.5 文件

C++ 中的文件处理类似于普通的交互输入和输出，因为用到了相同类型的流对象。从一个文件中的输入是由一个 ifstream 对象控制，它和由 istream 对象 cin 控制的从键盘输入的方式一样。与此类似，到一个文件中的输出是由一个 ofstream 对象控制，它和由 ostream 对象 cout 输出到显示器或打印机的方式一样。惟一的区别是 ifstream 和 ofstream 对象必须明确声明，并且用它们所控制的文件的外部名字初始化。还必须“# include”定义这些类的头文件（或者在前标准 C++ 中的 <fstream.h>）。

### 例 9.6 大写化一个文本文件中的所有单词

```
#include <fstream>
#include <iostream>
using namespace std;
int main ()
{
    ifstream infile ("input.txt");
    ofstream outfile ("output.txt");
    string word;
    char c;
    while (infile >> word)
    {
        if (word[0] >= 'a' && word[0] <= 'z') word[0] += 'A' - 'a';
        outfile << word;
        infile.get (c);
        outfile.put (c);
    }
}
```

图 9.4 展示了这个过程。

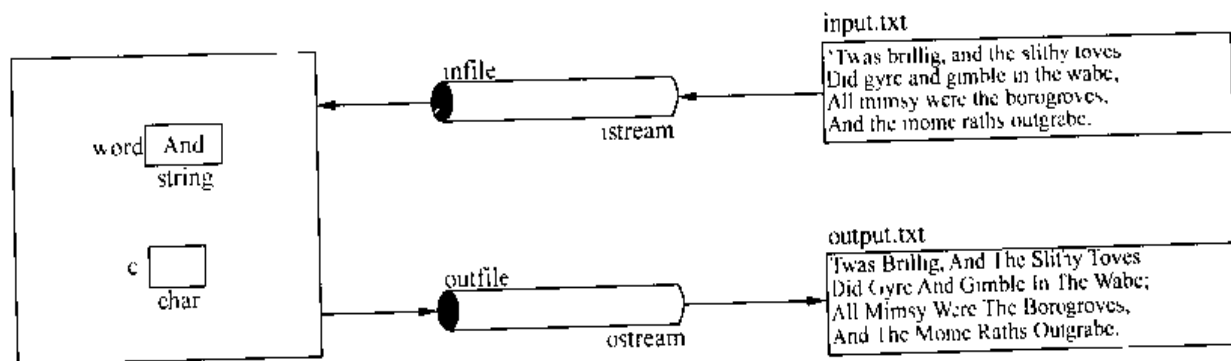


图 9.4 将字母变成大写示例

### 例 9.7 合并两个已排序的数据文件

这个程序把两个文件合成到第三个文件。存储在文件 `north.dat` 和 `south.dat` 中的成员按递增的顺序排列。程序同时读这两个输入文件并把他们所有数据复制到文件 `combined.dat` 中，使得他们都按递增的顺序排列。如图 9.5 所示。

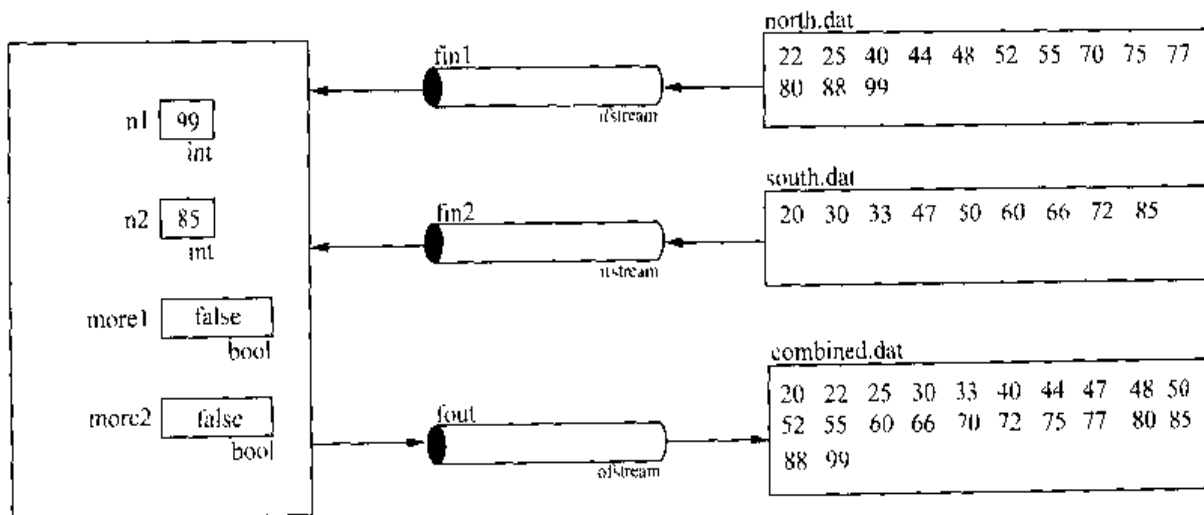


图 9.5 合并两个文件示例

```
bool more (ifstream& fin, int& n)
{ if (fin >> n) return true;
  else return false;
}

bool copy (ofstream& fout, ifstream& fin, int& n)
{ fout << " " << n;
  return more (fin, n);
}

int main ()
{ ifstream fin1 ("north.dat");
  ifstream fin2 ("south.dat");
  ofstream fout ("combined.dat");
  int n1, n2;
  bool more1 = more (fin1, n1);
  bool more2 = more (fin2, n2);
  while (more1 && more2)
    if (n1 < n2) more1 = copy (fout, fin1, n1);
    else more2 = copy (fout, fin2, n2);
  while (more1)
    more1 = copy (fout, fin1, n1);
  while (more2)
    more2 = copy (fout, fin2, n2);
  fout << endl;
}
```

`more ()` 函数用来从输入文件中读数据。每个调用都会尝试从文件 `fin` 中读一个整数给引用参数 `n`。如成功，返回 `true`，否则返回 `false`。`Copy ()` 函数把 `n` 的值写入到 `fout` 文件中，

然后, 调用 `more()` 函数从 `fin` 文件中读下一个整数给 `n`。当且仅当成功时, 它也返回 `true`。

前两次调用 `more()` 函数分别把 22 和 20 读到了 `n1` 和 `n2` 中。两次调用都返回 `true`, 这可以使主要的 `while` 循环开始。第一次重复时, 条件 (`n1 < n2`) 是错误的, 因此, `copy()` 函数从 `n2` 把 20 复制到 `combined.dat` 文件中, 然后又调用 `more()` 函数来把 30 读到 `n2` 中。第二次重复时, 条件 (`n1 < n2`) 是正确的 (因为 `22 < 30`), 因此, `copy` 函数从 `n1` 中把 22 复制到 `combined.dat` 文件中, 然后再调用 `more()` 函数把 25 读到 `n1` 中。再下次重复把 25 写到输出文件中, 然后把 40 读到 `n1` 中。再下次重复把 30 写到输出文件中, 然后把 33 读到 `n2` 中。这个过程一直持续下去, 直到把 85 从 `n2` 写到输出文件中, 再下次调用 `more()` 函数, 把 `false` 赋值给 `more2`。它中断了主要的 `while` 循环。然后第二个 `while` 循环重复三次, 在把 `more1` 设为 `false` 之前, 把最后三个整数从 `north.dat` 复制到 `combined.dat` 中。最后一次循环没有重复。

注意, 和其他的对象被传递的方式一样, 文件 (`fin1`, `fin2`, `fout`) 对象被传递给了函数。然而, 它们必须经常被引用传递。

## 9.6 字符串流

一个字符串流是一个流对象, 它允许字符串用做内在的文本文件。这叫做 `in-memory-I/O`。字符串流对缓冲输入和输出十分有用。在头文件 `<sstream>` 中有它们的类型 `istringstream` 和 `ostringstream` 的定义。

### 例 9.8 使用一个输出字符串流

这个程序创造了四个对象: 一个字符串 `s`, 一个整数 `n`, 一个浮点型数 `x` 和一个输出字符串流 `oss`, 如图 9.6 所示。

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
void print (ostringstream&);
int main ()
{ string s = "ABCDEFGH";
  int n = 33;
  float x = 2.718;
  ostringstream oss;
  print (oss);
  oss << s;
  print (oss);
  oss << " " << n;
  print (oss);
  oss << " " << x;
  print (oss);
}
```

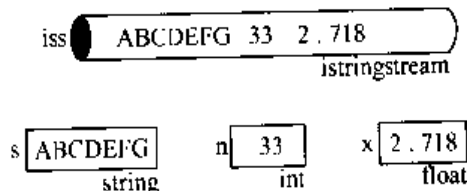


图 9.6 例 9.8 中的四个对象



```
void print (ostringstream& oss)
{ cout << "oss.str () = \"\" << oss.str () << "\"\" << endl";
}

oss.str () = ""
oss.str () = "ABCDEFGG"
oss.str () = "ABCDEFGG 33 "
oss.str () = "ABCDEFGG 33 2.718 "
```

输出字符串流对象 `oss` 的作用就像输出流对象 `cout`：字符串 `s` 的值，整数 `n` 和数 `x` 通过插入运算符 `<<` 的方式写给它。

一方面内在的对象 `oss` 像一个外部的文本文件，一方面它的内容可以作为一个 `string` 对象被调用 `iss.str()` 访问。

### 例 9.9 使用一个输入字符串流

程序和例 9.8 中的很相似，区别是它从一个输入字符串流 `iss` 读取，而不是写到一个输出字符串流。如图 9.7 所示。

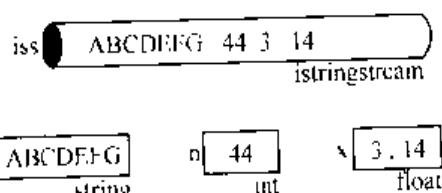


图 9.7 例 9.9 中的四个对象

```
void print (string& s, int, float, istringstream&);
int main ()
{ string s;
  int n = 0;
  float x = 0.0;
  istringstream iss ("ABCDEFGG 44 3.14");
  print (s, n, x, iss);
  iss >> s;
  print (s, n, x, iss);
  iss >> n;
  print (s, n, x, iss);
  iss >> x;
  print (s, n, x, iss);
}

void print (string& s, int n, float x, istringstream& iss)
{ cout << "s = \"\" << s << "\", n = \"\" << n << ", x = \"\" << x
  << ", iss.str () = \"\" << iss.str () << "\" << endl;
}

s = "", n=0, x = 0, iss.str () = "ABCDEFGG 44 3.14"
s = "ABCDEFGG", n=0, x = 0, iss.str () = "ABCDEFGG 44 3.14"
s = "ABCDEFGG", n=44, x = 0, iss.str () = "ABCDEFGG 44 3.14"
s = "ABCDEFGG", n=44, x = 3.14, iss.str () = "ABCDEFGG 44 3.14"
```

输入字符串流对象 `iss` 的作用就像输入流对象 `cin` 一样：即字符串 `s` 的值、整数 `n` 和数 `x` 通过取出运算符 `>>` 的方式从它读入。但 `iss` 对象也像一个外部文件：即从它读取但并不改变其内容。

## 复 习 题

### 9.1 一个 C 字符串和一个 C++ 字符串的区别是什么?

- 9.2 格式化输入和非格式化输入的区别是什么?
- 9.3 为什么空白部分不能用取出运算符读取?
- 9.4 流是什么?
- 9.5 C++ 如何简化字符串、外部文件和内部文件的处理?
- 9.6 有序访问与直接访问的区别是什么?
- 9.7 seekg () 和 seekp () 函数的功能是什么?
- 9.8 read () 和 write () 函数的功能是什么?

## 习 题

- 9.1 描述下面代码做了什么。

```
char cs1[] = "ABCDEFGHILJ";
char cs2[] = "ABCDEFGH";
cout << cs2 << endl;
cout << strlen(cs2) << endl;
cs2[4] = 'X';
if (strcmp(cs1, cs2) < 0) cout << cs1 << " < " << cs2 << endl;
else cout << cs1 " > " << cs2 << endl;
char buffer[80];
strcpy(buffer, cs1);
strcat(buffer, cs2);
char* cs3 = strchr(buffer, 'G');
cout << cs3 << endl;
```

- 9.2 描述下面代码做了什么。

```
string s = "ABCDEFGHIJKLMNO";
cout << s << endl;
cout << s.length() << endl;
s[8] = '!';
s.replace(8, 5, "xyz");
s.erase(6, 4);
cout << s.find("!");
cout << s.find("?");
cout << s.substr(6, 3);
s += "abcde";
string part(s, 4, 8);
string stars(8, '*');
```

- 9.3 描述当代码

```
string s;
int n;
float x;
cin >> s >> n >> x >> B;
```

对下面的每个输入执行时将发生什么情况:

- a. ABC 456 / .89 XYZ
- b. ABC 456/ .89 XYZ

c. ABC4567.89XYZ  
 d. ABC4567.89XYZ  
 e. ABC4567.89XYZ  
 f. ABC4567.89XYZ  
 g. ABC4567.89XYZ  
 h. ABC4567.89XYZ

9.4 对下面两个数据文件，跟踪例9.7中的合并程序。

north.dat

27 35 38 52 55 61 81 87

south.dat

31 34 41 45 49 56 63 74 92 95

9.5 写一个读取全名的程序，每行读取一个，然后以标准的电话目录的格式输出它们。例如，输入：

Johann Sebastian Bach  
 George Frederic Handel  
 George Philipp Emanuel Bach  
 Joseph Haydn  
 Johann Christian Bach  
 Wolfgang Amadeus Mozart

得到的输出如下：

Bach, Johann S.  
 Handel, George F.  
 Bach, Carl P.E.  
 Haydn, Joseph  
 Bach, Johann C.  
 Mozart, Wolfgang A.

9.6 写一个程序，对它输入中的行数、字数和字母出现频率统计数目并输出结果。例如，输入：

Two roads diverged in a yellow wood,  
 And sorry I could not travel both  
 And be one traveler, long I stood  
 And looked down one as far as I could  
 To where it bent in the undergrowth;

将会产生下面的输出：

The input had 5 lines, 37 words,  
 And the following letter frequencies:  
 A: 10 B: 3 C: 2 D: 13 E: 15 F: 1 G: 3 H: 4  
 I: 7 J: 0 K: 1 L: 8 M: 0 N: 12 O: 20 P: 0  
 Q: 0 R: 11 S: 5 T: 11 U: 3 V: 3 W: 6 X: 0  
 Y: 2 Z: 0

9.7 实现并测试下面的函数：

```
void reduce(string& s);
// 把s中的所有字母变为小写字母
// 并去掉所有从开始到结束的所有非字母
// 例子：如果 s = "Tis,"，那么 reduce(s) 把它变为"tis"
```

提示：首先写出并测试下面三个布尔型的函数：

```
bool is_uppercase(char c);
bool is_lowercase(char c);
bool is_letter(char c);
```

9.8 改变习题 9.6 的程序使得它计算出单词出现的频率而不是字母出现的频率。例如：输入

[I] then went to Wm. and Mary college, to wit in the spring of 1760, where I continued 2 years. It was my great good fortune, and what probably fixed the destinies of my life that Dr. Wm. Small of Scotland was then professor of Mathematics, a man profound in most of the useful branches of science, with a happy talent of communication, correct and gentlemanly manners, & an enlarged & liberal mind. He, most happily for me, became soon attached to me & made me his daily companion when not engaged in the school; and from his conversation I got my first views of the expansion of science & of the system of things in which we are placed.

将会产生输出：

The input had 8 lines and 120 words,  
with the following frequencies:

|                |                |                  |
|----------------|----------------|------------------|
| i: 3           | then: 2        | went: 1          |
| to: 3          | wm: 2          | and: 4           |
| mary: 1        | college: 1     | wit: 1           |
| in: 4          | the: 6         | spring: 1        |
| of: 11         | : 6            | where: 1         |
| continued: 1   | years: 1       | it: 1            |
| was: 2         | my: 3          | great: 1         |
| good: 1        | fortune: 1     | what: 1          |
| probably: 1    | fixed: 1       | destinies: 1     |
| life: 1        | that: 1        | dr: 1            |
| small: 1       | scotland: 1    | professor: 1     |
| mathematics: 1 | a: 2           | man: 1           |
| profound: 1    | most: 2        | useful: 1        |
| branches: 1    | science: 2     | with: 1          |
| happy: 1       | talent: 1      | communication: 1 |
| correct: 1     | gentlemanly: 1 | manner: 1        |
| an: 1          | enlarged: 1    | liberal: 1       |
| mind: 1        | he: 1          | happily: 1       |
| for: 1         | me: 3          | became: 1        |
| soon: 1        | attached: 1    | made: 1          |
| his: 2         | daily: 1       | companion: 1     |
| when: 1        | not: 1         | engaged: 1       |
| school: 1      | from: 1        | conversation: 1  |
| got: 1         | first: 1       | views: 1         |
| expansion: 1   | system: 1      | things: 1        |
| which: 1       | we: 1          | are: 1           |
| placed: 1      |                |                  |

9.9 写一个右对齐文本的程序。它应该重复读入一个左对齐的行序列，然后以右对齐的格

式输出它们。例如，输入：

```
Listen, my children, and you shall hear
Of the midnight ride of Paul Revere,
On the eighteenth of April, in Seventy - five;
Hardly a man is now alive
Who remembers that famous day and year.
```

将会输出为：

```
Listen, my children, and you shall hear
    Of the midnight ride of Paul Revere,
On the eighteenth of April, in Seventy - five;
                        Hardly a man is now alive
Who remembers that famous day and year.
```

### 9.10 实现并测试下面的函数。

```
string Roman (int n);
// 返回等价于阿拉伯数字的罗马数字
// 数字 n
// 前提: n > 0, n < 3888
// 例如: Roman (1776) 返回 "MDCCCLXXVI"
// Roman (1812) 返回 "MDCCCXII", Roman (1945) 返回 "MCMXLV"
```

### 9.11 实现并测试下面的函数。

```
int HinduArabic (string s);
// 返回等价于罗马数字的阿拉伯数字
// 数字在字符串 s 中给定
// 前提: s 包含一个正确的罗马数字
// 例如: HinduArabic ("MDCCCLXXVI") 返回 1776;
// HinduArabic ("MDCCCXII") 返回 1812
```

### 9.12 实现附录 G 中的算法 G.1，把十进制数转换为十六进制的数。

```
string hexadecimal (int n);
// 返回表示 n 的十六进制数
// 前提: n >= 0
// 后续条件: 在返回字符串中的每个字符是一个十六进制的数，并且那个字符串是等于 n 的十六进制数
// 11643 返回 16 进制数 2d7b
```

[提示：写一个辅助函数 char c (int k)，对十六进制的数 k 返回十六进制的字符。例如，c (14) 返回 'e']

### 9.13 实现附录 G 中的算法 G.2，把十六进制数转换为十进制数。

```
string decimal (int n);
// 返回表示十六进制的十进制数
// 数字存储在字符串 s 中
// 前提: s.length() > 0 并且每个 s[i] 是一个十六进制数
// 后续条件: 返回值是十进制的等价值
// "2d7b" 返回 10 进制数 11643
```

注意，这是习题 9.12 中的 hexadecimal () 函数的反向。[提示：写一个辅助函数 int v (string s, int i)，对十六进制字符 s[i] 返回十进制数。例如：v ("2d7b", 3)

返回 12。]

#### 9.14 实现并测试下面的函数。

```
void reverse (string& s);
// 把字符串 s 反向
// 后续条件: s[i] < - - > s[len-i-1]
// 例如: reverse(s) 把 s = "ABCDEFGH" 变为 "HGFEDCBA"
[提示: 使用一个临时字符串。]
```

#### 9.15 实现并测试下面的函数:

```
bool is_palindrome (string s);
// 如果 s 是一个回文 (指顺读和倒读都一样的词语), 那么返回 true
// 例如: is_palindrome("RADAR") 返回 true,
// is_palindrome("ABCD") 返回 false
```

#### 9.16 更改 253 页中的程序程序 9.7, 使得它合并下面开始部分显示的两个名字的已排序文件, 把得到的排序的行输出到一个名字为 Presidents.dat 的文件和一个 cout 文件。

[提示: 使用 getline(fin, s)。]

republicans

```
Bush, George Herbert Walker
Coolidge, Calvin
Eisenhower, Dwight David
Ford, Gerald Rudolph
Harding, Warren Gamaliel
Hoover, Herbert Clark
McKinley, William
Nixon, Richard Milhous
Reagan, Ronald Willson
Roosevelt, Theodore
Taft, William Howard
```

Democrats

```
Carter, James Earl
Clinton, William Jefferson
Johnson, Lyndon Baines
Kennedy, John Fitzgerald
Roosevelt, Franklin
Truman, Harry S
Wilson, Woodrow
```

## 复习题答案

- 9.1 一个 C 字符串就是一个字符的排列, 它使用非空字符 '\0' 来标志字符的结束。一个 C++ 字符串是一个对象, 它的 string 类型在 <string> 文件中定义并且它有一个很大的函数指令系统, 例如 length() 和 replace()。

```
char cs[8] = "ABCDEFGH"; // cs 是一个 C 字符串
string s = "ABCDEFGH"; // s 是一个 C++ 字符串
cout << s << " has " << s.length() << " characters. \n";
s.replace(4, 2, "yz"); // 把 s 变为 "ABCDyzG"
```

- 9.2 格式化输入使用取出运算符来忽略空白部分。非格式化输入使用 get() 函数和 getline() 函数。get() 函数读入在输入流中的下一个字符, 它不忽略空白部分。getline() 函数读入在输入流中的字符的所有剩余部分, 直到它到达换行符 '\n', 然后取出换行符并忽略它。

- 9.3 空白部分（空格、标号、换行等）不能用取出运算符读，因为它忽略了所有的空白部分。
- 9.4 一个流就是一个在程序和数据源之间控制输出和输出的对象。C++ 允许 `<iostream>` 对象面向交互式输入和输出 I/O（即，`cin` 和 `cout`），允许 `<fstream>` 对象面向外部文件和 `<sstream>` 对象面向内部文件（字符串流）。
- 9.5 C++ 对所有字符串、外部文件、内部文件定义相同系列的函数和操作，藉此来简化对它们的处理。例如，取出运算符对从键盘、从一个外部文件和一个字符串流输入一个 `double` 的作用是一样的。
- 9.6 有序访问必须从开始部分开始，按顺序一个接一个地访问每个元素。直接访问允许直接访问每个元素，它通过由它的下标数或地址来定位。数组允许直接访问。磁带只有有序访问，但 CD 可以直接访问。如果在一个火车上，从一个车厢到另一个车厢必须使用有序访问。但开始上火车时却是直接访问。直接访问比有序访问快，但是它要求一些外部机制（如数组下标、文件字节数、铁路站台）。
- 9.7 为了允许直接的访问，在一个外部文件中 `seekg()` 函数和 `seekp()` 函数分别定位 `get` 指针和 `put` 指针。例如，调用 `input.seekg(24)` 把 `get` 指针定位于文件中的字节数 24，这个文件绑定在名字为 `input` 流的文件中。
- 9.8 `read()` 函数和 `write()` 函数分别用来直接访问外部文件的输入和输出。例如，调用 `input.read(s.c_str(), n)` 将从文件中直接复制 `n` 个字节到字符串 `s` 中，这个文件绑定在名字为 `input` 的文件流中。

## 习题答案

- 9.1
- ```
char cs1[ ] = "ABCDEFGHILJ";           // 定义 cs1 为 C 字符串
char cs2[ ] = "ABCDEFGH";             // 定义 cs2 为 C 字符串
cout << cs2 << endl;                  // 输出: ABCDEFGH
cout << strlen(cs2) << endl;          // 输出: 8
cs2[4] = 'X';                         // 把 cs2 变为 ABCDXFGH
if (strcmp(cs1, cs2) < 0) cout << cs1 << " < " << cs2 << endl;
else cout << cs1 << " > = " << cs2 << endl;

char buffer[80];                      // 定义 buffer 为一个小于 80 个字符的 C 字符串
strcpy(buffer, cs1);                  // 把 buffer 变为 ABCDEFGHILJ
strcat(buffer, cs2);                  // 把 buffer 变为 ABCDEFGHILJABCDXFGH
char* cs3 = strchr(buffer, 'G');      // 让 cs3 指向 buffer[6]
cout << cs3 << endl;                  // 输出: GHIJABCDXFGH
```
- 9.2
- ```
string s = "ABCDEFGHILJKLMNOP";       // 定义 s 为这个字符串
cout << s << endl;                    // 输出: ABCDEFGHILJKLMNOP
cout << s.length() << endl;           // 输出: 16
s[8] = '!';                           // 把 s 变为 "ABCDEFGH!JKLMNOP"
s.replace(10, 5, "xyz");               // 把 s 变为 "ABCDEFGH! JxyzP"
s.erase(2, 4);                        // 把 s 变为 "ABGH! JxyzP"
```

```

cout << s.find("!") << endl           // 输出: 4
cout << s.find("?") << endl           // 输出: 10
cout << s.substr(3, 6) << endl         // 输出: H! Jxyz
s += "abode";                          // 把 s 变为"ABGH! JxyzPabode"
string part(s, 1, 10);                  // 定义 part 为"BCH! JxyzPa"
string stars(8, '*');                   // 定义 stars 为"*****"

```

### 9.3 a. ABC 456 7.89 XYZ

把 ABC 赋给了 s, 把 456 赋给了 n, 把 7.89 赋给了 x, 然后把 "XYZ" 赋给了 s。

### b. ABC 4567 .89 XYZ

把 ABC 赋给了 s, 把 4567 赋给了 n, 把 0.89 赋给了 x, 然后把 "XYZ" 赋给了 s。

### c. ABC 456 7.8 9XYZ

把 ABC 赋给了 s, 把 456 赋给了 n, 把 7.8 赋给了 x, 然后把 "9XYZ" 赋给了 s。

### d. ABC456 7.8 9 XYZ

把 ABC456 赋给了 s, 然后崩溃了, 因为 7.8 不是一个合法的整型文字。

### e. ABC456 7 .89 XYZ

把 ABC456 赋给了 s, 把 7 赋给了 n, 把 0.89 赋给了 x, 然后把 "XYZ" 赋给了 s。

### f. ABC4 56 7.89XY Z

把 ABC4 赋给了 s, 把 56 赋给了 n, 然后崩溃了, 因为 7.8XY 不是一个合法的 float 型文字。

### g. AB C456 7.89 XYZ

把 AB 赋给了 s, 然后崩溃了, 因为 C456 不是一个合法的整型文字 (注意十六进制数 c456, 它也可写为 C456, 它作为一个合法的整型文字是合格的。但作为输入, 十六进制必须加上前缀 "0x", 如 0xc456)。

### h. AB C 456 7.89XYZ

把 AB 赋给了 s, 然后崩溃了, 因为 C 不是一个合法的整型文字。

### 9.4 跟踪合并程序:

| n1 | n2 | more1 | more2 |
|----|----|-------|-------|
| 27 | 31 | true  | true  |
| 35 |    |       |       |
|    | 34 |       |       |
|    | 41 |       |       |
| 38 |    |       |       |
| 52 |    |       |       |
|    | 45 |       |       |
|    | 49 |       |       |
|    | 56 |       |       |
| 55 |    |       |       |
| 61 |    |       |       |



(续表)

| n1 | n2 | more1 | more2 |
|----|----|-------|-------|
| 81 | 63 | false |       |
|    | 74 |       |       |
|    | 92 |       |       |
| 87 | 95 |       | false |

```

9.5 int main ()
{
    string word, first, last;
    char c;
    bool is_first, is_last = true;
    string name [32];
    int n = 0;
    while (cin >> word)
    {
        cin.get (c);          // 应该是一个空格或者是一个换行
        is_first = is_last;    // 当前的单词是第一个名字
        is_last = bool (c == '\n'); // 当前的单词是最后一个名字
        if (is_first) first = word;
        else if (is_last) name [n++] = word + ", " + first;
        else first += " " + word.substr (0, 1) + "."; // 增加初始化
    }
    --n;
    for (int i = 0; i < n; i++)
        cout << '\t' << i+1 << ". " << name [i] << endl;
}

```

```

9.6 int main ()
{
    string word;
    const int SIZE = 90; // 对频率数组 (int ('Z') == 90)
    int lines = 0, words = 0, freq [SIZE] = {0}, len;
    char c;
    while (cin >> word)
    {
        ++words;
        cin.get (c);
        if (c == '\n') ++lines;
        len = word.length ();
        for (int i = 0; i < len; i++)
        {
            c = word [i];
            if (c >= 'a' && c <= 'z') c += 'A' - 'a'; // 大写 c
            if (c >= 'A' && c <= 'Z') ++freq [c]; // 计数 c
        }
        cout << "The input had " << lines << " lines, " << words
            << " words, \n and the following letter frequencies: \n";
        for (int i = 65; i < SIZE; i++)
        {
            cout << '\t' << char (i) << ": " << freq [i];
            if (i > 0 && i % 8 == 0) cout << endl; // 把8输出到一行
        }
        cout << endl;
    }
}

```

```

9.7 bool is_upper (char c)
    { return bool (c >= 'A' && c <= 'Z');
    }

    bool is_lower (char c)
    { return bool (c >= 'a' && c <= 'z');
    }

    bool is_letter (char c)
    { return bool (is_upper (c) || is_lower (c));
    }

    void reduce (string& s)
    { while (s.length () > 0 && ! is_letter (s [0]))
        s.erase (0, 1);
      int k = s.length () - 1;
      while (k > 0 && ! is_letter (s [k-- ]))
        s.erase (k+1, 1);
      int len = s.length ();
      if (len == 0) return;
      for (int i=0; i<len; i++)
        if (is_upper (s [i])) s [i] += 'a' - 'A';
    }

9.8 int main ()
    { ifstream in ("Pr0907.in");
      string s;
      const int SIZE=1000;      // 假设最多 1000 个单词
      string word [SIZE];      // 记录读入的词
      int lines=0, words=0, n=0, freq [SIZE] = {0}, i;
      char c;
      while (in >> s)
        { reduce (s);
          if (s.length () == 0) continue;
          ++words;
          in.get (c);
          if (c == '\n') ++lines;          // 计数行数
          for (i=0; i<n; i++)
            if (word [i] == s) break;
          if (i == n) word [n++] = s;      // 把 word 加到列表
          ++freq [i];                     // 计数单词
        }
      cout << "The input had " << lines << " lines and " << words
        << " words, \n with the following frequencies: \n";
      for (int i=0; i<n; i++)
        { s = word [i];
          if (i > 0 && i%3 == 0) cout << endl;      // 把 3 输出到 一行
          cout << setw (16) << setiosflags (ios:: right)
            << s.c_str () << ": " << setw (2) << freq [i];
        }
      cout << endl;
    }

9.9 int main ()
    {

```

```

const int SIZE=100;    // 存储的最大行数
string line [SIZE], s;
int n=0, len, maxlen=0;
while (! cin.eof ())
{
    getline (cin, s);
    len = s.length ();
    if (len > 0) cout << s << endl;
    if (len > maxlen) maxlen = len;
    line [n++] = s;
}
// n 等于读取的行数
for (int i=0; i<n; i++)
{
    s = line [i];
    len = s.length ();
    cout << string (maxlen-len, ' ') << s << endl;
}
}

```

9.10 string Roman (int n)

```

{
    int d3 = n/1000;    // 以千计的数字
    string s (d3, 'M');
    n %= 1000;
    int d2 = n/100;    // 以百计的数字
    if (d2 == 9) s += "CM";
    else if (d2 >= 5)
    {
        s += "D";
        s += string (d2-5, 'C');
    }
    else if (d2 == 4) s += "CD";
    else s += string (d2, 'C');
    n %= 100;
    int d1 = n/10;    // 以十计数的数字
    if (d1 == 9) s += "XC";
    else if (d1 >= 5)
    {
        s += "L";
        s += string (d1-5, 'X');
    }
    else if (d1 == 4) s += "XI";
    else s += string (d1, 'X');
    n %= 10;
    int d0 = n/1;    // 个位数字
    if (d0 == 9) s += "IX";
    else if (d0 >= 5)
    {
        s += "V";
        s += string (d0-5, 'I');
    }
    else if (d0 == 4) s += "IV";
    else s += string (d0, 'I');
    return s;
}

```

9.11 int v (string s, int i)

```

{
    char c = s [i];
    if (c == 'M') return 1000;
}

```

```

    if (c == 'D') return 500;
    if (c == 'C') return 100;
    if (c == 'L') return 50;
    if (c == 'X') return 10;
    if (c == 'V') return 5;
    if (c == 'I') return 1;
    return 0;
}

int HindArabic (string s)
{ int n0 = 0, n1 = 0, n = 0;
  for (int i = 0; i < s.length (); i++)
  { n0 = n1;
    n += r1 = v (s, i);
    if (n1 > n0) n -= 2 * n0;
  }
  return n;
}

```

```

9.12 char c (int k)
{ assert (k >= 0 && k <= 15);
  if (k < 10) return char (k + '0');
  return char (k - 10 + 'a');
}

string hexadecimal (int n)
{ if (n == 0) return string (1, '0');
  string s;
  while (n > 0)
  { s = string (1, c (n%16)) + s;
    n /= 16;
  }
  return s;
}

```

```

9.13 int v (string s, int i)
{ char c = s [i];
  assert (c >= '0' && c <= '9' || c >= 'a' && c <= 'f');
  if (c >= '0' && c <= '9') return int (c - '0');
  else return int (c - 'a' * 10);
}

int decimal (string s)
{ int len = s.length ();
  assert (len > 0);
  int n = 0;
  for (int i = 0; i < len; i++)
    n = 16 * n + v (s, i);
  return n;
}

```

```

9.14 void reverse (string& s)
{ string temp = s;
  int len = s.length ();
  for (int i = 0; i < len; i++)
    s [i] = temp [len - i - 1];
}

```

```

9.15 bool is_palindrome (string s)

```

```

{ int len = s.length ();
  for (int i = 0; i < len/2; i++)
    if (s[i] != s[len-i-1]) return false;
  return true;
}

```

9.16

```

bool more (ifstream& fin, string& s)
{ if (getline (fin, s)) return true;
  else return false;
}

```

```

bool copy (ofstream& fout, ifstream& fin, string& s)
{ fout << s << endl;
  cout << s << endl;
  return more (fin, s);
}

```

```

int main ()
{ ifstream fin1 ("Democrats.dat");
  ifstream fin2 ("Republicans.dat");
  ofstream fout ("Presidents.dat");
  string s1, s2;
  bool more1 = more (fin1, s1);
  bool more2 = more (fin2, s2);
  while (more1 && more2)
    if (s1 < s2) more1 = copy (fout, fin1, s1);
    else more2 = copy (fout, fin2, s2);
  while (more1)
    more1 = copy (fout, fin1, s1);
  while (more2)
    more2 = copy (fout, fin2, s2);
  fout << endl;
}

```

# 第 10 章 类

## 10.1 引言

类就像一个排列：它是由其他不同类型的元素派生出来的一种类型。但它又不像一个排列，一个类的元素可以有不同的类型。而且，类的元素可以是函数，包括运算符。

存储器的每层一般都可以称为一个“对象”，但这个词通常是用来描述类型是类的变量的。因此，“面向对象的编程”包括了用类去编程。把一个完备的实体称为一个对象，它有自己的数据和函数。一个对象“知道”如何处理它自己的事情，它的这种机能使它看起来好像有生命力。

面向对象的程序设计并不仅仅是简单地用类去编程。然而，这只是万里长征的第一步，还有许多规则需要足够的重视。

## 10.2 类的声明

下面是一个类的声明，它的对象表示有理数（例如分数）。

```
class Ratio
{ public:
    void assign (int, int);
    double convert ();
    void invert ();
    void print ();
private:
    int num, den;
};
```

声明是由关键词 `class` 开始的，紧接着是类的名字并按要求的分号结束。这个类的名字是 `Ratio`。

函数 `assign ()`、`convert ()`、`invert ()` 和 `print ()` 叫做成员函数，因为它们是类的成员。类似地，变量 `num` 和 `den` 叫做数据成员。成员函数也称做方法和服务。

在这个类中，所有的成员函数都设计成了 `public` 型的，所有的数据成员都设计成了 `private` 型的。区别是：`public` 型的成员从类的外部可以直接访问，而 `private` 型的成员只限于类的内部进行访问。拒绝从类的外部进行访问称做“信息隐蔽”。这可以让程序设计人员把软件模块化，从而使软件更易于理解、调试和维护。

下面的例子显示了如何实现和应用类。

### 例 10.1 类 Ratio 的实现

```
class Ratio
{
public:
    void assign (int, int);
    double convert ();
    void invert ();
    void print ();
private:
    int num, den;
};

int main ()
{
    Ratio x;
    x.assign (22, 7);
    cout << "x = ";
    x.print ();
    cout << " = " << x.convert () << endl;
    x.invert ();
    cout << "1/x = ";    x.print ();
    cout << endl;
}

void Ratio:: assign (int numerator, int denominator)
{
    num = numerator;
    den = denominator;
}

double Ratio:: convert ()
{
    return double (num) /den;
}

void Ratio:: invert ()
{
    int temp = num;
    num = den;
    den = temp;
}

void Ratio:: print ()
{
    cout << num << '/' << den;
}

x = 22/7 = 3.14286
1/x = 7/22
```

这里的 `x` 是类 `Ratio` 声明的一个对象。因此它有自己的数据成员 `num` 和 `den`，它可以调用四个成员函数 `assign ()`、`convert ()`、`invert ()` 和 `print ()`。注意：一个成员函数如 `invert ()` 被调用时在它名字的前面加前缀，前缀是拥有这个函数的对象的名字：`x.invert ()`。事实上，一个成员函数也只有用这种方式调用。我们称对象 `x` “拥有”这个调用。

对象  $x$  的声明就如一个变通的变量一样，它的类型是 `Ratio`。可以把这个类型称为“用户定义的类型”。把新的 `Ratio` 类型加入到以前定义的数值型 (`int`, `float` 等等) 队列中去，C++ 允许对程序设计语言定义的扩展。可以把对象  $x$  想像成如图 10.1 所示的样子。

注意标识符 `Ratio::`，作为一个前缀作用于每一个函数名。这对于在类的外部定义每个成员函数很有必要。生存空间解析运算符 `::` 用来把每个函数的定义约束在类 `Ratio` 上。没有这个标识符，编译器就不会知道所定义的函数是类 `Ratio` 的成员函数。如例 10.2 所示，通过把成员函数的定义包括在声明的内部来避免发生这种情况。

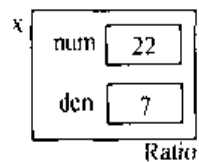


图 10.1 对象  $x$  示例

如例 10.1 所示，如果 `Ratio` 声明了对象  $x$  的话，称类已被实例化，并称这个对象为类的一个实例。就像同一种类型可以有許多变量一样，同一个类也可以有許多实例，例如：

```
Ratio x, y, z;
```

### 例 10.2 类 `Ratio` 的一个完备的实现

下面是同一个类 `Ratio`，其中成员函数的定义包含在了类的声明中。

```
class Ratio
{
public:
    void assign (int n, int d) { num = n; den = d; }
    double convert () { return double (num) /den; }
    void invert () { int temp = num; num = den; den = temp; }
    void print () { cout << num << '/' << den; }
private:
    int num, den;
};
```

大多情况下，定义成员函数的首选类型是在类声明的外部，像例 10.1 那样用生存空间解析运算符来定义。这种方式自然地把成员函数的声明与它的定义分开，与通常的信息隐蔽的规则相一致。实际上，成员函数的定义经常放在一个文件中并单独编译。问题是使用了这个类的应用程序只需知道对象能做什么；它们不需要知道对象如何做。函数的声明告诉了它们做什么，函数的定义讲述了如何实现它。当然以前定义的类型 (`int`, `double` 等等) 也是这么实现的：当用一个浮点型的去除另一个时，我们知道结果应该是什么，但并不知道除法是怎么实现的（也就是应用了什么算法）。尤为重要的是，我们并不想知道。如果不得不考虑这些细节，那将会把注意力从手边的任务中分开。这种观点一般就叫做“信息隐蔽”，在面向对象的程序设计中这是一条很重要的原则。

像例 10.1 中所示的那样，当成员函数的定义与它的声明是分开的时候，声明部分叫做类的接口，包含成员函数定义的那部分叫做类的实现。程序设计人员在用类时，接口是必须考虑的一部分。实现则隐蔽在一个独立的文件中，因此“隐蔽”了用户（也就是编程人员）不必知道的信息。类的实现部分经常由实现者来完成，它们独立于程序设计人员，但程序设计人员将会用到他们实现的类。



## 10.3 构造函数

例 10.1 中定义类 `Ratio` 用 `assign ( )` 函数来初始化它的对象。如果在对象声明时进行初始化可能会更自然些。普通（预定义）类型是这样做的，示例如下：

```
int n = 22;
char * s = "Hello";
```

C++ 允许类的对象用构造函数来进行简单方式的初始化。

构造函数是这样的一个成员函数，当对象声明时它也就自动生成。构造函数必需和类自身的名字一样，并且声明时不返回什么类型。下面的例子讲述了如何用一个构造函数来替代 `assign ( )` 函数。

### 例 10.3 类 `Ratio` 的一个构造函数

```
class Ratio
{
public:
    Ratio (int n, int d) { num = n; den = d; }
    void print () { cout << num << "/" << den; }
private:
    int num, den;
};

int main ()
{
    Ratio x (-1, 3), y (22, 7);
    cout << "x = ";
    x.print ();
    cout << " and y = ";
    y.print ();
}

x = -1/3 and y = 22/7
```

构造函数和例 10.1 中的 `assign ( )` 函数具有同样的效果：它通过赋给它的数据成员特定的值来初始化对象。当 `x` 的声明实现时，它会自动地调用构造函数，这样整型的 `-1` 和 `3` 就传给了它的参数 `n` 和 `d`。然后函数把这些值赋给了 `x` 的数据成员 `num` 和 `den`。因此语句

```
Ratio x (-1, 3), y (22, 7);
```

就等价于下面三行：

```
Ratio x, y;
x.assign (-1, 3);
y.assign (22, 7);
```

构造函数为对象分配和初始化存储空间，并且完成这个函数设计的其他任何任务，通过这样来“构造”类的对象。从字面上说，它从一大堆无用的位中创建一个活生生的对象。

可以形象地描述一下类 Ratio 和它的初始化了的对象的关系, 如图 10.2 所示。

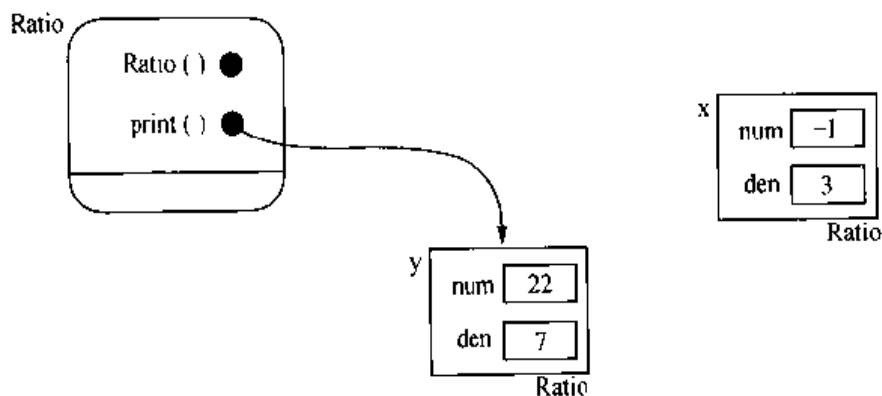


图 10.2 Ratio 类与其实例的关系

类本身用一个圆框来表示, 其中包含它的成员函数。每个函数包含一个叫做“this”的指针, 它指向调用它的对象。这里的图形表示了当程序的最后一行执行时的状况, 这时对象 y 正调用 print() 函数: y.print()。此时, 构造函数的“this”指针是空的, 因为它没有被调用。

一个类可以有几个构造函数。就像其他任何重载函数一样, 可以根据不同的参数表来区别它们。

#### 例 10.4 在 Ratio 类中增加构造函数

```
class Ratio
{
public:
    Ratio() { num = 0; den = 1; }
    Ratio(int n) { num = n; den = 1; }
    Ratio(int n, int d) { num = n; den = d; }
    void print() { cout << num << "/" << den; }
private:
    int num, den;
};
```

```
int main()
{
    Ratio x, y(4), z(22, 7);
    cout << "x = ";
    x.print();
    cout << "\ny = ";
    y.print();
    cout << "\nz = ";
    z.print();
}
```

```
x = 0/1
y = 4/1
z = 22/7
```

这里的类 Ratio 有三个构造函数。第一个没有参数, 它用默认的 0 和 1 初始化已声明的对象。第二个构造函数有一个整型的参数, 并把这个整数赋给对象的一个参数来初始化对

象。

在类的不同类型构造函数中,最简单的是这种不带参数的类型,称做默认的构造函数。如果构造函数在类的定义中没有明确声明,系统会自动为类生成它。正如例 10.1 中所述的那样。

## 10.4 构造函数初始化列表

大部分构造函数只是对对象的数据成员进行初始化。因此, C++ 为构造函数提供了一种特殊的语法设备来简化代码,这个设备就是初始化列表。

用初始化列表重写例 10.2 中的第三个构造函数如下:

```
Ratio (int n, int d) : num (n), den (d) { }
```

在函数体中,省去了把 `n` 赋给 `num` 和 `d` 赋给 `den` 的赋值语句。替代它们的黑体字中的初始化列表。注意列表是由冒号开始,而函数体现在是空的。

下面是用初始化列表来重写 `Ratio` 的三个构造函数。

### 例 10.5 在类 `Ratio` 中使用初始化列表

```
class Ratio
{ public:
    Ratio () : num (0), den (1) { }
    Ratio (int n) : num (n), den (1) { }
    Ratio (int n, int d) : num (n), den (d) { }
private:
    int num, den;
};
```

当然,这三个独立的构造函数并不是必要的。如下面的例子所描述的那样,可以用默认的参数值来把它们合成一个简单的构造函数。

### 例 10.6 在类 `Ratio` 的构造函数中使用默认的参数值

```
class Ratio
{ public:
    Ratio (int n=0, int d=1) : num (n), den (d) { }
private:
    int num, den;
};

int main ()
{ Ratio x, y (4), z (22, 7);
}
```

这里, `x` 代表  $0/1$ , `y` 代表  $4/1$ , `z` 代表  $22/7$ 。

记住,只有在实际的参数没有传递时才使用默认的参数值。因此,在类 `Ratio` 中,对象

x 没有值传过来，0 赋值给了正式的参数 n，然后，再赋值给 x.num；1 赋值给了正式的参数 n，然后再赋值给 x.den。在对象 y 的声明中，只有 4 传递过来，然后它赋值给了 y.num，正式的参数 d 赋值为 1，它赋值给了 y.den。对象 z 的声明中没有用到默认的参数值。

## 10.5 访问函数

类的数据成员通常声明为 private 型的，以此来限制对它的访问。但通常也包含 public 型的成员函数来对它进行只读访问。这样的函数叫做访问函数。（在 Java 中，它们叫做 getty 方法，因为在它们的名字中经常用到“get”。正好相反的是 setty 方法，它用来改变数据成员的值，并且名字中经常用到“set”。getty 方法是只读的，setty 方法是只写的。）

### 例 10.7 类 Ratio 中的访问函数

```
class Ratio
{ public:
    Ratio (int n=0, int d=1) : num (n), den (d) , {
        int numerator () const { return num; }
        int denominator () const { return den; }
    private:
        int num, den;
    };

    int main ()
    { Ratio x (22, 7);
      cout << x.numerator () << '/' << x.denominator () << endl;
    }
```

函数 numerator ( ) 和 denominator ( ) 返回 private 型的数据成员值。

注意，在这两个存取函数的声明中关键词 const 的应用。函数可以象常量对象一样被应用（见 10.9 节）

## 10.6 私有成员函数

类的数据成员通常声明为 private 型的，成员函数通常声明为 public 型的。但这种二分法并不是必须的。在有些情况下，把一个或更多的成员函数声明为 private 型的也很有益处。如下所述，这些函数只能被类本身应用。也就是说，它们是局部效用函数。

### 例 10.8 使用私有型的成员函数

```
class Ratio
{ public:
    Ratio (int n=0, int d=1) : num (n), den (d) { reduce (); }
    void print () const { cout << num << '/' << den << endl; }
```

```

private:
    int num, den;
    void reduce ();
};

int gcd (int, int);
void Ratio:: reduce ()
{ // 强制不变量 (den > 0):
    if (num == 0 || den == 0)
    { num = 0;
      den = 1;
      return;
    }
    if (den < 0)
    { den *= -1;
      num *= -1;
    }
    // 强制不变量 (gcd (num, den) == 1):
    if (den == 1) return; // 它已经简化了
    int sgn = (num < 0? -1: 1); // gcd () 非负
    int g = gcd (sgn * num, den);
    num /= g;
    den /= g;
}

int gcd (int m, int n)
{ // 返回 m, n 的最大公约数
    if (m < n) swap (m, n);
    while (n > 0)
    { int r = m % n;
      m = n;
      n = r;
    }
    return m;
}

int main ()
{ Ratio x (22, 7);
  cout << x.numerator () << '/' << x.denominator () << endl;
}
-5/18

```

这个例子包含了 private 型的函数 reduce ()，它用 gcd () 函数（见问题 5.18）来把分数 num/den 减化为最简单的形式。因此分数 100/-360 被存为 -5/18。

除了有 - 一个单独的 reduce () 函数外，可以在构造函数内做这个实际的减法。但这样做有两个好的理由。把函数构造和做减法组合在一起会违反软件规则：独立的任务应当由独立的函数来处理。此外，reduce () 函数在后面要用来简化 Ratio 对象进行的数学操作的结果。

注意关键词 public 和 private 叫做访问标识符；它们指定是否在类定义的外部可以访问它的成员。关键词 protected 是第三个访问标识符。它会在第 13 章中描述到。

## 10.7 复制构造函数

每个类至少有两个构造函数。这可以根据他们特殊的声明判断出来。

```
X ();           //默认的构造函数
X (const X&);   //复制的构造函数
```

此时的 X 就是标识符。例如，类 Widget 的两个特殊的构造函数可以声明如下：

```
Widget ();           //默认的构造函数
Widget (const Widget&); //复制的构造函数
```

这两个特殊的构造函数中的第一个叫做默认的构造函数，当一个对象以最简单的形式声明如下时：

```
Widget x;
```

它自动调用默认的构造函数。第二个叫做复制的构造函数，当一个对象被如下复制时：

```
Widget y (x);
```

它自动地调用复制的构造函数。如果这两个中的任一个都没有明确定义，系统会含蓄地自动定义它。

注意复制的构造函数带有一个参数：它要复制的对象。这个对象由常数参数传递，因为它不应当变化。

当复制构造函数被调用时，它把已存在的一个对象的全部状态复制到同一个类的一个新的对象上。如果类的定义没有明确地包含一个复制构造函数（就像前面的例子中所有的都没有那样），系统会默认地自动生成一个。具有写自己的复制构造函数的能力可以使软件更加容易控制。

### 例 10.9 为类 Ratio 加一个复制构造函数

```
class Ratio
{
public:
    Ratio (int n=0, int d=1) : num (n), den (d) { reduce (); }
    Ratio (const Ratio& r) : num (r.num), den (r.den) { }
    void print () { cout << num << "/" << den; }
private:
    int num, den;
    void reduce ();
};

int main ()
{
    Ratio x (100, 360);
    Ratio y (x);
    cout << "x = ";
    x.print ();
    cout << ", y = ";
    y.print ();
}
```

```
|
x = 5/18, y = 5/18
```

复制构造函数把参数 *r* 中的数据成员 *num* 和 *den* 复制到构造的对象中。当 *y* 被声明时，它调用复制构造函数把 *x* 复制到 *y*。

注意复制构造函数所要求的语法：它只能有一个参数，参数和声明是同一个类，并且它必须由常数参数 `const X&` 传递。

复制构造函数在下列情况下被自动调用：

- 当一个对象以声明初始化的方式复制时；
- 当一个对象作为值传递给一个函数时；
- 当一个对象作为值从一个函数返回时。

### 例 10.10 跟踪对复制构造函数的调用

```
class Ratio
{ public:
    Ratio (int n=0, int d=1) : num (n), den (d) { reduce (); }
    Ratio (const Ratio& r) : num (r.num), den (r.den)
        { cout << "COPY CONSTRUCTOR CALLED \n"; }
private:
    int num, den;
    void reduce ();
};
```

```
Ratio f (Ratio r)    // 调用复制构造函数，把 r 复制到 r
{ Ratio s = r;       // 调用复制构造函数，把 r 复制到 s
  return s;          // 调用复制构造函数，把 s 复制到 ?
}
```

```
int main ()
{ Ratio x (22, 7);
  Ratio y (x);       // 调用复制构造函数，把 x 复制到 y
  f (y);
}
```

```
COPY CONSTRUCTOR CALLED
COPY CONSTRUCTOR CALLED
COPY CONSTRUCTOR CALLED
```

在这个例子中，复制构造函数被调用四次。当 *y* 声明时调用它，把 *x* 复制成 *y*；当 *y* 作为值传递给函数 *f* 时调用它，把 *y* 复制成 *r*；当 *s* 声明时调用它，把 *r* 复制成 *s*；当函数 *f* 作为值返回时调用它，这时甚至什么都没复制。注意 *s* 的初始化看起来像是赋值，但作为声明的一部分，它像 *y* 一样调用了复制构造函数。

当类定义中没有包含一个复制构造函数时，编译器会自动地产生一个。这种默认的复制构造函数将会按位地简单复制对象。在大多数情况下，这也正是所需要的。因此在这些情况下，一个明确定义的复制构造函数是没有必要的。

然而，在一些重要的情况下，一个按位的复制并不够。第 9 章中描述的字符串就是一个

很好的例子。在那个类的对象中，相关的数据成员只是对实际的字符串拥有一个指针，因此，一个按位的复制只能复制指针，而不是字符串本身。像这种情况，就很有必要定义自己的复制构造函数。

## 10.8 类的析构函数

当一个对象创建时，它就自动调用构造函数来维持它的生命。类似地，当一个对象的生命结束时，它就调用另一个专门的成员函数来结束它的生命。这个函数叫做析构函数。

每个类只有一个析构函数。如果在类的定义中没有明确定义，像默认的构造函数，复制的构造函数和赋值运算符一样，析构函数会自动创建。

### 例 10.11 在类 Ratio 中包含一个析构函数

```
class Ratio
{
public:
    Ratio () { cout << "OBJECT IS BORN. \n"; }
    ~Ratio () { cout << "OBJECT DIES. \n"; }
private:
    int num, den;
};

int main ()
{
    Ratio x; // x 范围的开始
    cout << "Now x is alive. \n";
    | // x 范围的结束
    cout << "Now between blocks. \n";
    | Ratio y;
    cout << "Now y is alive. \n";
    |
}
```

```
OBJECT IS BORN.
Now x is alive.
OBJECT DIES.
Now between blocks.
OBJECT IS BORN.
Now y is alive.
OBJECT DIES.
```

这里是当构造函数和析构函数调用时的显示结果。

当它到达它的使用范围末尾时，对象调用类析构函数。对于一个局部对象，这也是它声明中的模块结束的时候。对于一个静态对象，则是在 main ( ) 函数的末尾。

尽管系统会自动创建它们，但应当养成一个好的程序设计习惯来为每个类定义复制构造函数、赋值运算符和析构函数。



## 10.9 常量对象

如果它不应改变的话, 创建一个对象常量是一个好的程序设计习惯。这可以通过关键词 `const` 来实现:

```
const char BLANK = ' ';
const int MAX_INT = 2147483647;
const double PI = 3.141592653589793;
void init (float a [ ], const int SIZE);
```

像变量和函数参数一样, 对象也可以声明为常量:

```
const Ratio PI (22, 7);
```

然而, 如果这样做的话, C++ 编译器会限制对这个对象的成员函数的访问。例如, 如果类 `Ratio` 以前已经定义, 这个对象就不能调用 `print ()` 函数。

```
//PI.print (); //错误: 不允许调用
```

实际上, 除非更改类的定义, 能被 `const` 对象调用的成员函数只能是构造函数和析构函数。为了克服这种限制, 必须把能用 `const` 对象调用的那些成员函数声明为常量。

函数可以通过在它的参数表和函数体间插入关键词 `const` 来声明。例如:

```
void print () const { cout << num << '/' << endl; }
```

函数定义的改变将允许常量对象对它的调用。例如:

```
const Ratio PI (22, 7);
PI.print (); // 现在可以了
```

## 10.10 结构体

C++ 类是 C 中结构体的一般形式, 结构体是一个只有成员没有函数的类。类由于它的成员函数而有了生命力, 并且由于其私有型的数据成员而拥有了信息的隐蔽, 人们通常地把类看做结构体。

为了保持与旧的 C 语言的一致性, C++ 保留了 `struct` 关键字对结构体进行定义。然而, 一个 C++ 结构体与一个 C++ 类基本相同。C++ 结构体与 C++ 类的惟一显著区别在于默认的赋予成员的访问标识符。尽管不是特别推荐地, C++ 类可以定义而不必要明确地指定它的成员访问标识符。例如:

```
class Ratio
{ int num, den;
};
```

是 `Ratio` 类的一种合法的定义。由于数据成员 `num` 和 `den` 的访问标识符没有指定, 它被默认的设为私有型的。如果用结构体替代类, 则如下所示:

```
struct Ratio
{
    int num, den;
};
```

则数据成员被默认地设为公有型的。但这可以通过指定访问标识符来很简单地改变。例如：

```
struct Ratio
{
    private:
        int num, den;
};
```

因此类和 C++ 结构体的区别其实很微小。

## 10.11 对象的指针

在很多应用程序中，使用对象（和结构体）的指针很方便。下面是个简单的例子。

### 例 10.12 应用类的指针

```
class X
{
    public:
        int data;
};

int main ()
{
    X* p = new X;
    (*p).data = 22;           // 等价于: p->data = 22;
    cout << " (*p).data = " << (*p).data << " = " << p->data << endl;
    p->data = 44;
    cout << " p->data = " << (*p).data << " = " << p->data << endl;
}

(*p).data = 22 = 22
p->data = 44 = 44
```

因为  $p$  是  $X$  对象的一个指针， $*p$  就是一个  $X$  对象， $(*p).data$  访问了它的 `public` 型数据成员。注意在表达式  $(*p).data$  中括号是必须的，因为直接的成员选择运算符“.”比非关联运算符“ $*$ ”具有较高的优先级（见附录 C）。

以下两种表示法具有同样的意义：

```
(*p).data
p->data
```

当用到指针时，“箭头”符号“ $\rightarrow$ ”较为优先选择，因为它简单明了，并且表示了“ $p$  指向的东西”。

下面是一个更为重要的例子。

### 例 10.13 链表的一个 Node 类

```
class Node
```

```

: public:
    Node (int d, Node* q=0) : data (d), next (q) {}
    int data;
    Node* next;
};

```

它定义了一个 Node 类，类的对象包含一个整型数据成员和一个 next 指针。

```

int main ()
{ int n;
  Node* p;
  Node* q=0;
  while (cin >> n)
  { p = new Node (n, q);
    q = p;
  }
  for ( ; p; p = p->next )
    cout << p->data << " -> ";
  cout << "*" << "\n";
}

```

22 33 44 55 66 77 \*  
77 -> 66 -> 55 -> 44 -> 33 -> 22 \*

首先注意 Node 类的定义中包含了两个对类自身的引用。这是允许的，因为每个引用实际上是一个指向类的指针。也要注意构造函数初始化了数据成员。

这个程序允许用户创建一个反向链表。然后它遍历链表，输出每个 data 值。

While 循环连续读入整数给 n，直到用户输入文件结束符 (Ctrl+D)。在循环内，程序得到一个新结点，插入整数到它的数据成员，并把新结点连接到前面的结点（由 q 指向的）。最后，for 循环由 p 指向的结点开始遍历这个链表（构造的最后一个结点），直到 p 是空的。

这个例子中构造的链表可以形象化表示如图 10.3 所示。

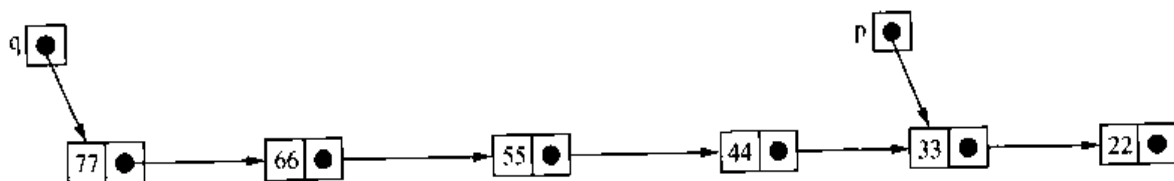


图 10.3 例 10.3 中的链表

## 10.12 静态数据成员

有时，一个数据成员的简单值适用于类的所有成员。在这种情况下，在类的每个对象中都存储同样的值就会效率不高。把数据成员声明为静态的就能避免这种情况。做法是在变量的声明开始时把关键字 static 包含进去。它要求这个变量定义成全局的。因此语法看起来如下：

```

class X
{ public:
    static int n; //n 声明为了一个静态数据成员
}

```

```
};
int X::n = 0;    // n 的定义
```

静态变量自动初始化为 0，因此在定义中明确的初始化并不必要，除非想让它有一个非零的初始值。

### 例 10.14 一个静态数据成员

Widget 类包含了一个静态的数据成员 count，它记录了全局存在的 Widget 对象的数量。每次创建一个 Widget 时（由构造函数）计数器增 1，每次摧毁一个 Widget（由析构函数），计数器减 1。

```
class Widget
{ public:
    Widget () { ++count; }
    ~Widget () { --count; }
    static int count;
};

int Widget::count = 0;

int main ()
{ Widget w, x;
  cout << "Now there are " << w.count << " widgets. \n";
  { Widget w, x, y, z;
    cout << "Now there are " << w.count << " widgets. \n";
  }
  cout << "Now there are " << w.count << " widgets. \n";
  Widget y;
  cout << "Now there are " << w.count << " widgets. \n";
}

Now there are 2 widgets.
Now there are 6 widgets.
Now there are 2 widgets.
Now there are 3 widgets.
```

注意在内部模块四个 Widget 是如何创建的，然后当程序离开那个模块的时候它们被摧毁，把 Widget 的全局数量从 6 减少到 2。

一个静态的数据成员就像一个普通的全局变量：不管这个类有多少个实例，只存在这个变量的一个复制。主要的差别在于它是类的一个数据成员，因此可能是私有型的。

### 例 10.15 私有型的静态数据成员

```
class Widget
{ public:
    Widget () { ++count; }
    ~Widget () { --count; }
    int numWidgets () { return count; }
private:
    static int count;
```

```

};
int Widget::count = 0;

int main ()
{
    Widget w, x;
    cout << "Now there are " << w.runWidgets () << " widgets. \n";
    } Widget w, x, y, z;
    cout << "Now there are " << w.numWidgets () << " widgets. \n";
    }
    cout << "Now there are " << w.runWidgets () << " widgets. \n";
    Widget y;
    cout << "Now there are " << w.numWidgets () << " widgets. \n";
    }
}

```

这个例子和例 10.2 的作用一样。但现在静态的变量 `count` 是私有型的，在 `main ()` 中需要访问函数 `numWidget ()` 来读它。

类、类的成员、类的对象的关系可以形象化表示为图 10.4 所示的样子。

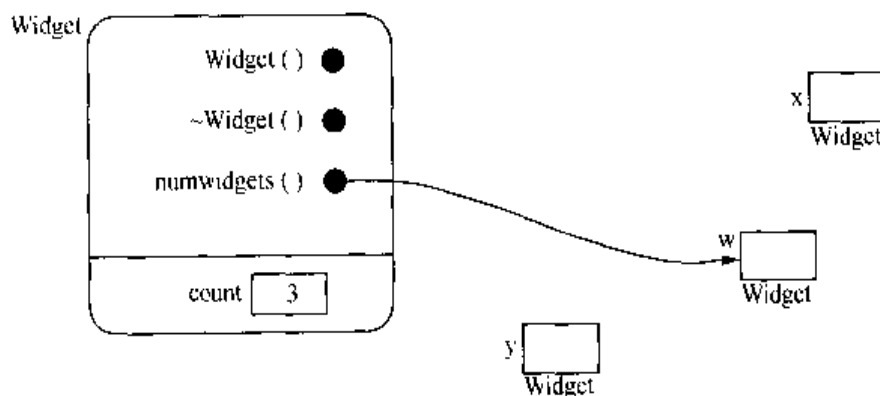


图 10.4 类、成员与对象的关系

圆形框表示类本身，它包含三个成员函数和数据成员 `count`。公有型成员在线的上部，私有型成员在下面。每个成员函数包括一个指向属于当前函数调用的对象的指针（叫做“this”）。一个简单的印象表明了程序的最后一行执行期间的状态：存在三个 `Widget`（`w`、`x` 和 `y`），`w` 正在调用 `numWidgets ()` 函数，它返回私有型数据成员 `count` 的值。注意这个数据成员留在类本身；类对象没有数据。

### 10.13 静态函数成员

像任何普通的成员函数一样，例 10.2 中的 `numWidgets ()` 函数要求它应该是类的一些实例所拥有的。但由于它返回独立于个体对象自身的静态数据成员 `count` 的值，它并不介意哪个对象调用它。每次让 `w` 调用它，但当 `x`、`y`、`z` 存在时，也能让 `x` 或 `y` 或 `z` 调用它。此外，除非已创建了一些对象，根本不能调用它。这相当武断。由于函数的行为独立于实际的函数对象，最好是让调用也独立于它们。这可以简单地通过声明函数为静态的来实现。

### 例 10.16 一个静态函数成员

Widget 类包含了一个静态的数据成员 count，它记录了全局存在的 Widget 对象的数量。每次创建一个 Widget 时（由构造函数）计数器增 1，每次摧毁一个 Widget（由析构函数）计数器减 1。

```
class Widget
{
public:
    Widget () { ++count; }
    ~Widget () { --count; }
    static int num () { return count; }
private:
    static int count;
};

int Widget::count = 0;

int main ()
{
    cout << "Now there are " << Widget::num () << " widgets. \n";
    Widget w, x;
    cout << "Now there are " << Widget::num () << " widgets. \n";
    {
        Widget w, x, y, z;
        cout << "Now there are " << Widget::num () << " widgets. \n";
    }
    cout << "Now there are " << Widget::num () << " widgets. \n";
    Widget y;
    cout << "Now there are " << Widget::num () << " widgets. \n";
}
```

把 numWidgets () 函数声明为静态的使它独立于类的实例。因此现在它是作为 Widget 类的一个成员使用生存空间解析运算符“::”调用的。允许这个函数在任何对象实例化之前被调用。

前一个图形显示了类中的关系，它的实例现在应该看起来如图 10.5 所示。

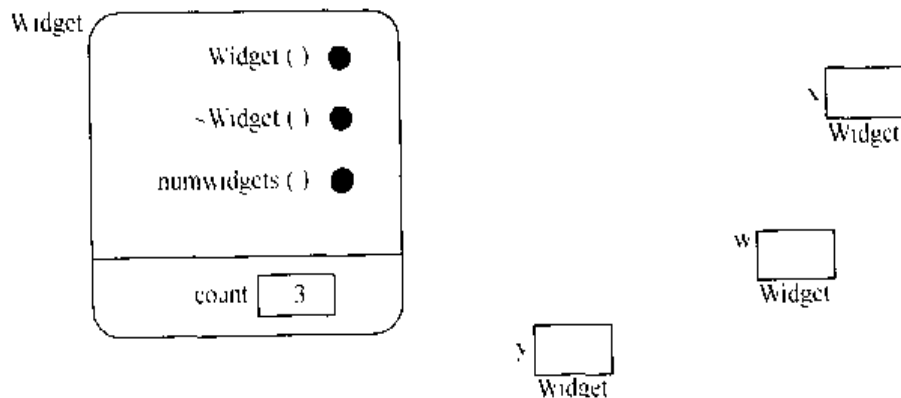


图 10.5 新的类、成员及对象的关系

区别在于现在成员函数 num () 没有了“this”指针。作为一个静态成员函数，它和类

本身相联系，而不是和它的实例相联系。

静态成员函数只能访问它们自己类中的静态数据。

## 复 习 题

- 10.1 说明一个类的公有成员和私有成员的区别。
- 10.2 说明一个类的接口和实现的区别。
- 10.3 说明一个类的成员函数和应用函数的区别。
- 10.4 说明构造函数和析构函数的区别。
- 10.5 说明默认的构造函数和其他构造函数的区别。
- 10.6 说明复制构造函数和赋值运算符的区别。
- 10.7 说明访问函数和效用函数的区别。
- 10.8 说明在 C++ 中类和结构体的区别。
- 10.9 构造函数必须有什么名字？
- 10.10 析构函数必须有什么名字？
- 10.11 一个类可以有多少个构造函数？
- 10.12 一个类可以有多少个析构函数？
- 10.13 生存空间解析运算符:: 在类定义中如何使用？为什么使用？
- 10.14 如果在类定义中没有被编程人员包括在内，则哪些成员函数是自动生成的？
- 10.15 在下面的代码中复制构造函数被调用多少次？

```
Widget f (Widget u)
{
    Widget v (u);
    Widget w = v;
    return w;
}

main ()
{
    Widget x;
    Widget y = f (f (x));
}
```

- 10.16 为什么在表达式 (\*p).data 中需要圆括号？

## 习 题

- 10.1 实现一个三维点 (x, y, z) 的 Point 类。包含一个构造函数，一个复制构造函数，一个把这个点变成负的函数 negate ()，一个返回到 (0, 0, 0) 点的距离的函数 norm () 和一个 print () 函数。
- 10.2 实现一个整数栈的类 Stack 类。包含一个默认的构造函数，一个析构函数和普通的栈

操作: push ()、pop ()、isEmpty () 和 isFull ()。使用数组实现。

- 10.3 实现一个 Time 类。这个类的每个对象将表示一天中的一个特殊时刻, 把小时、分钟和秒存储为整型。包含一个构造函数, 访问函数, 一个提前已知对象的当前时间的函数 advance (int h, int m, int s), 一个重新设置已知对象当前时间的函数 reset (int h, int m, int s) 和一个 print () 函数。
- 10.4 实现一个 Random 类, 产生伪随机数。
- 10.5 实现一个 Person 类。这个类的每个对象将表示一个人。数据成员将包括人的名字、生日和死亡日期。包含一个默认的构造函数, 一个析构函数、访问函数和一个打印函数。
- 10.6 实现一个 String 类。这个类的每个对象将表示一个字符串。数据成员是这个串的长度和实际的字符串。此外还有构造函数、析构函数、访问函数和一个打印函数。还包括一个“下标”函数。
- 10.7 为  $2 \times 2$  复数实现一个 Matrix 类:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

包括一个默认的构造函数, 一个复制构造函数, 一个返回矩阵的逆的 inverse () 函数, 一个返回矩阵行列式的 det () 函数, 一个布尔型函数 isSingular (), 它根据行列式是否为 0 返回 1 或 0, 以及一个 print () 函数。

- 10.8 为二维点 (x, y) 实现一个 Point 类。包括一个默认的构造函数, 一个复制构造函数, 一个把这个点转换成它的负数的 negate () 函数, 一个返回这个点到原点 (0, 0) 距离的 norm () 函数和一个 print () 函数。
- 10.9 实现一个 Circle 类。这个类的每个对象将表示一个圆, 存储它的半径和它的中心的坐标 x 和 y。包含一个默认的构造函数, 访问函数, 一个 area () 函数和一个 circumference () 函数。

## 复习题答案

- 10.1 一个公有成员从类的外部可以访问; 一个私有成员则不然。
- 10.2 类的接口由数据成员和成员函数原型 (也就是只是函数的声明) 组成。类的实现包括成员函数的定义。
- 10.3 类的成员函数是类的一部分, 因此它访问类的 private 部分。一个应用函数在类的外部声明, 因此它不能访问类的 private 部分。
- 10.4 一个构造函数是类的一个成员函数, 当这个类的对象实例化时 (也就是构造时), 它自动执行。一个析构函数是一个类的成员函数, 当对象的范围结束时 (也就是被摧毁时), 它自动执行。
- 10.5 默认的构造函数是独一无二的构造函数, 它没有参数 (或者是一个参数都是默认值的构造函数)。



- 10.6 当类的对象被除了直接的赋值外的任何机制复制时，类的一个复制构造函数执行。这包括初始化，作为值传递一个参数，和作为值返回。
- 10.7 一个访问函数就是一个公有型函数，它返回类的数据成员中的一个的值。一个效用函数是一个私有型函数，它只是在类的内部用来执行“技术的”任务。
- 10.8 在 C++ 中一个类和一个结构体本质上是相同的。惟一显著的区别就是类的默认访问层是私有型的，而结构体的默认访问层是公有型的。
- 10.9 每个类的构造函数必须和类本身具有相同的名字。
- 10.10 每个类的析构函数必须和类本身具有相同的名字，由一个颞化符号（~）做前缀。
- 10.11 一个类可以有多少个构造函数，没有限制。但由于多重的构造函数是函数重载，它们都要由它们的参数列表来辩认。
- 10.12 一个类只能有一个析构函数。
- 10.13 生存空间解析运算符::通常用来“解决外部的调用”。当一个成员函数的定义在类定义范围的外部时，它就用在类的定义中。
- 10.14 在类定义中如果它们没有由编程人员包含进去，共有四个类的成员函数由编译器自动生成：默认的构造函数、复制构造函数、析构函数和重载的赋值运算符。
- 10.15 在这段代码中复制构造函数被调用 7 次。每次对函数 f 的调用要求 3 次对复制构造函数的调用：当参数作为值传给 u 时，当 v 初始化时，当 w 作为值返回时。第 7 次调用是初始化 y 时。
- 10.16 在表达式 (\*p).data 中圆括号是必须的，因为直接的成员选择运算符“.”比解除参照运算符“\*”有更高的优先级。

## 习题答案

- 10.1 一个 Point 类的实现使用了普通的方式：以一个底线结束每个数据成员。这有个好处，匹配构造函数参数（x、y 和 z）和它们相应的数据成员（x\_、y\_和 z\_）很容易，而不会有冲突。

```
#include <cmath>
#include <iostream>
using namespace std;
class Point
{ public:
    Point (float x=0, float y=0, float z=0): x_ (x), y_ (y), z_ (z) {}
    Point (const Point& p): x_ (p.x_), y_ (p.y_), z_ (p.z_) {}
    void negate () { x_ *= -1; y_ *= -1; z_ *= -1; }
    double norm () { return sqrt (x_*x_ + y_*y_ + z_*z_); }
    void print ()
        { cout << " (" << x_ << ", " << y_ << ", " << z_ << ")"; }
private:
    float x_, y_, z_;
};
```

- 10.2 在这个 Stack 类的实现中, top 总是栈中栈顶元素的索引。数据成员 size 是容纳栈元素数组的大小。因此, 当它包含这个数目的元素时, 栈是满的。构造函数把 size 的默认设为 10。

```
class Stack
{ public:
    Stack (int s=10) : size (s), top (-1) { a = new int [size]; }
    ~Stack () { delete [] a; }
    void push (const int& item) { a [ ++top ] = item; }
    int pop () { return a [ top-- ]; }
    bool isEmpty () const { return top == -1; }
    bool isFull () const { return top == (size-1); }
private:
    int size;    // 数组大小
    int top;     // 栈顶
    int * a;     // 容纳栈元素的数组
};
```

### 10.3 class Time

```
{ public:
    Time (int h=0, int m=0, int s=0)
        : hr (h), min (m), sec (s) { normalize (); }
    int hours () { return hr; }
    int minutes () { return min; }
    int seconds () { return sec; }
    void advance (int h=0, int m=0, int s=1);
    void reset (int h=0, int m=0, int s=0);
    void print () { cout << hr << ":" << min << ":" << sec; }
private:
    int hr, min, sec;
    void normalize ();
};

void Time::normalize ()
{ min += sec/60;
  hr += min/60;
  hr %= 24;
  min %= 60;
  sec %= 60;
}

void Time::advance (int h, int m, int s)
{ hr += h;
  min += m;
  sec += s;
  normalize ();
}

void Time::reset (int h, int m, int s)
{ hr = h;
  min = m;
  sec = s;
  normalize ();
}
```

10.4 一个 `Random` 类的实现使用了一个效用函数 `normalize()`，它标准化 `Time` 对象，使得它的三个数据成员在正确的范围内： $0 \leq \text{sec} < 60$ ， $0 \leq \text{min} < 60$ ， $0 \leq \text{hr} < 24$ 。它也使用了效用函数 `randomize()`，实现了 1949 年由 D.H. Lehmer 提出的 Linear Congruential 算法。效用函数 `_next()` 通过调用 `_randomize()` 随机数次来更新 `_seed()`。

```
#include <climits>      // 定义 INT_MAX 和 ULONG_MAX 常量
#include <ctime>         // 定义 time() 函数
#include <iomanip>        // 定义 setw() 函数
#include <iostream>      // 定义 cout 对象
using namespace std;

class Random
{ public:
    Random (long seed=0) { _seed = ( seed ? seed : time (NULL) ); }
    void seed (long seed=0) { _seed = ( seed ? seed : time (NULL) ); }
    int integer () { return _next (); }
    int integer (int min, int max)
        { return min + _next () % (max - min + 1); }
    double real ()
        { return double (_next ()) / double (INT_MAX); }
private:
    unsigned long _seed;
    void _randomize ()
        { _seed = (314159265 * _seed + 13579) % ULONG_MAX; }
    int _next ()
        { int iterations = _seed % 3;
          for (int i=0; i <= iterations; i++) _randomize ();
          return int (_seed/2); }
};

int main ()
{ Random random;
  for (int i = 1; i <= 10; i++)
      cout << setw (16) << setiosflags (ios:: right)
          << random.integer ()
          << setw (6) << random.integer (1, 6)
          << setw (12) << setiosflags (ios:: fixed | ios:: left)
          << random.real () << endl;
}
```

测试驱动程序对三个随机数函数的每一个调用了 10 次，产生了 10 个范围在 0 到 2 147 483 647 之间的伪随机整数，10 个范围在 1 到 6 之间的伪随机整数和 10 个范围在 0.0 到 1.0 之间的伪随机实数。

#### 10.5 class Person

```
{ public:
    Person (char * _n=0, int _b=0, int _y=0):
        _Person () { delete _n; name_ = _n; }
    char * name () { return name_; }
    int born () { return yob_; }
```

```

    int died () { return yod ; }
    void print ();
private:
    int len ;
    char * name _;
    int yob_, yod_;
};

Person:: Person (const char * name, int yob, int yod)
    : len_ (strlen (name)),
      name_ (new char [len_ + 1]),
      yob_ (yob),
      yod_ (yod)
{ memcpy (name_, name, len_ + 1);
}

void Person:: print ()
{ cout << " \tName: " << name_ << endl;
  if (yob_) cout << " \tBorn: " << yob_ << endl;
  if (yod_) cout << " \tDied: " << yod_ << endl;
}

```

为了保持对象是独立的, `name` 存储为一个单独的字符串。为了方便它的单独存储, 把它的长度存储在了数据成员 `len_` 中, 使用 `memcpy ()` 函数 (定义在 `string.h` 中) 来把字符串 `name` 复制到字符串 `name_` 中。然后析构函数使用 `delete` 运算符来释放这次存储。

- 10.6 一个 `String` 类的实现包括三个构造函数: 带有可选的参数 `size` 的默认的构造函数, 一个允许一个对象用一个普通的 C 字符串初始化的构造函数, 还有复制的构造函数。第二个访问函数叫做 `convert ()`, 因为它实际上转换类型 `String` 到 `char *` 类型。“下标”函数叫做 `character ()`, 因为它返回字符串中的一个字符——由参数 `i` 索引的那一个。

```

class String
{ public:
    String (short = 0);           // 默认的构造函数
    String (const char *);       // 构造函数
    String (const String&);      // 复制构造函数
    ~String () { delete [] data; } // 析构函数
    int length () const { return len; } // 访问函数
    char * convert () { return data; } // 访问函数
    char character (short i) { char c = data [i]; return c; }
    void print () { cout << data; }
private:
    short len;           // 字符串的字符数 (非空的)
    char * data;         // 字符串
};

String:: String (short size) : len (size)
{ data = new char [len + 1];
  for (int i = 0; i < len; i++) data [i] = ' ';
  data [len] = '\0';
}

String:: String (const char * str) : len (strlen (str))
{ data = new char [len + 1];
}

```

```

        memcpy (data, str, len+1);

String::String (const String& str) : len (str.len)
{ data = new char [len+1];
  memcpy (data, str.data, len+1);
}

```

### 10.7 class Matrix

```

{ public:
    Matrix (double a=0, double b=0, double c=0, double d=0)
        : a_ (a), b_ (b), c_ (c), d_ (d) {}
    Matrix (const Matrix& m) :
        a_ (m.a_), b_ (m.b_), c_ (m.c_), d_ (m.d_) {}
    double det () { return a_*d_ - b_*c_; }
    int isSingular () { return det () == 0; }
    Matrix inverse ();
    void print ();
private:
    double a_, b_, c_, d_;
};

Matrix Matrix::inverse ()
{ double k = 1/det ();
  Matrix temp (k*d_, -k*b_, -k*c_, k*a_);
  return temp;
}

void Matrix::print ()
{ cout << a_ << " " << b_ << "\n" << c_ << " " << d_ << "\n";
}

```

### 10.8 class Point

```

{ public:
    Point () : _x (0.0), _y (0.0) {}
    Point (double x, double y) : _x (x), _y (y) {}
    Point (const Point& p) : _x = p._x; _y = p._y; {}
    double norm () const { return sqrt (_x*_x+_y*_y); }
    void print () const
        { cout << " (" << _x << ", " << _y << ")" ; }
    void negate () { _x = -1.0*_x; _y = -1.0*_y; }
private:
    double _x;
    double _y;
};

```

### 10.9 class Circle

```

{ public:
    Circle () : _x (0.0), _y (0.0), _radius (1.0) {}
    Circle (float x, float y, float radius)
        : _x (x), _y (y), _radius (radius) {}
    Circle (const Circle& C)
        { _x = C._x; _y = C._y; _radius = C._radius; }
    float diameter () const { return 2.0 * _radius; }
    float area () const
        { return 3.141592654 * _radius * _radius; }
    float circumference () const

```

```
        { return PI * diameter (); }  
void print () const  
{ cout << "Center is at (" << _x << ", " << y  
  << ") and " << "Radius = " << _radius ;  
private;  
  float _x;  
  float _y;  
  float _radius;  
};
```

# 第 11 章 重载运算符

## 11.1 引言

C++ 包含有丰富的 45 个运算符的存储。在附录 C 中有它们的摘要。这些运算符被自动定义成基本的类型（int，float 等）。当定义一个类时，实际上也创建了一个类型。C++ 的大部分运算符都可以重载以适应新类的类型。这一章描述如何实现它。

## 11.2 赋值运算符重载

在所有的运算符中，赋值运算符“=”可能用得最多。它的目的是复制一个对象到另一个对象。像默认的构造函数、复制构造函数和析构函数，赋值运算符在定义的每个类中都自动生成。但也像这三种其他的成员函数一样，赋值运算符在类的定义中也可以明确定义。

### 例 11.1 为类 **Ratio** 加一个赋值运算符

下面是类 Ratio 的接口，显示了默认的构造函数，复制的构造函数和赋值运算符：

```
class Ratio
{
public:
    Ratio (int = 0, int = 1);           //默认的构造函数
    Ratio (const Ratio&);               //复制的构造函数
    void operator = (const Ratio&);     //赋值运算符
    // 其他的声明
private:
    int num, den;
};
```

注意赋值运算符的必要语法。成员函数的名字是 operator =，它的参数表和复制构造函数的一样：它包含一个同类的简单参数，由常量参数传递过来的。

下面是重载运算符的实现：

```
void Ratio::operator = (const Ratio& r)
{
    num = r.num;
    den = r.den;
}
```

它简单地把对象 r 的成员数据复制到属于对它调用的对象。

### 11.3 this 指针

C++ 允许链式赋值，如下：

```
x = y = z = 3.14;
```

它执行时先把 3.14 赋值给 z，然后是 y，最后是 x。但像例 11.1 所显示的那样，赋值运算符实际上是一个命名为 operator= 的函数。在这个链中，这个函数被调用三次。第一次调用时，它把 3.14 赋给了 z，因此函数的输入是 3.14。在第二次调用时，它把 3.14 赋给了 y，因此它的输入值也必须是 3.14。因此这个值应该是第一次调用的输出（也就是返回值）。类似地，第二次调用的输出应该又是 3.14，并作为第三次调用的输入。这个函数的三次调用是嵌套的，如下：

```
f(x, f(y, f(z, 3.14)))
```

问题是赋值运算符是个应当返回它赋的值的函数。所以，不返回类型 void，赋值运算符应当返回一个和赋值的对象相同的类型。例如：

```
Ratio& operator = (const Ratio& r)
```

它允许链式赋值

#### 例 11.2 重载的赋值运算符的较易接受的原型

```
class Ratio
{
public:
    Ratio (int = 0, int = 1);           //默认构造函数
    Ratio (const Ratio&);              //复制构造函数
    Ratio& operator = (const Ratio&);  //赋值运算符
    //其他的声明
private:
    int num, den;
    //其他的声明
};
```

类 T 中一个重载赋值运算符的比较流行的语法是：

```
T& operator = (const T&);
```

返回类型是同一个类 T 的对象的引用。但是，在赋值时，函数应当按顺序返回正被赋值的对象，这种方式有很大的束缚。因此，当赋值运算符正被重载为一个成员函数时，它应当返回调用它的对象。既然这个宿主对象没有其他命名的变量，C++ 定义了一个特殊的指针，命名为 this，来指向这个宿主对象。

可以想像 this 指针如图 11.1 所示。

下面给出重载赋值运算符的正确实现。



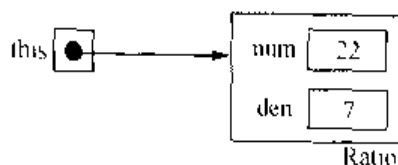


图 11.1 this 指针示例

### 例 11.3 类 Ratio 的赋值运算符的实现

```
Ratio& Ratio::operator= (const Ratio& r)
{
    num = r.num;
    den = r.den;
    return *this;
}
```

现在 Ratio 类可以链式赋值：

```
Ratio x, y, z (22, 7);
x = y = z;
```

类 T 的一个重载赋值运算符的正确实现是：

```
T& T::operator= (const T& r)
    //把 t 的每个成员数据赋给相应的宿主的成员数据
    return *this;
}
```

最后，注意到赋值和初始化是不同的，尽管它们都有相同的符号：

```
Ratio x (22, 7); //初始化
Ratio y (x);     //初始化
Ratio z = x;     //初始化
Ratio w;
w = x;           //赋值
```

初始化调用复制构造函数。赋值调用赋值运算符。

## 11.4 算术运算符重载

所有的程序设计语言都提供了数值型的标准算术运算符加、减、乘和除。因此，为用户定义的数值类型，如类 Ratio 定义这些就是很自然的事了。旧的程序设计语言如 C 和 Pascal，可以按如下的方式来定义函数：

```
Ratio product (Ratio x, Ratio y)
{
    Ratio z (x.num*y.num, x.den*y.den);
    return z;
}
```

这很不错。但函数不得不以传统的方式调用：

```
z = product (x, y);
```

C++ 允许应用标准的算术运算符符号来定义这种函数，因此它们可以更加自然地调用：

```
z = x * y;
```

像 C++ 中的大多运算符一样，乘法运算符有一个函数名，它用了指定的单词 `operator`，名字是 “`operator*`”。用它来代替上面代码中的 “`product`”，通常希望重载函数像下面这样：

```
Ratio operator* (Ratio x, Ratio y)
{ Ratio z (x.num * y.num, x.den * y.den);
  return z;
}
```

但它并不是一个成员函数。如果是，将不得不把它改为一个参数。这个 `operator*` 函数有两个参数。

由于重载的算术运算符不能是成员函数，它们也不能访问私有的数据成员 `num` 和 `den`。幸运地是，C++ 允许这个规则有个例外，因此能完成重载的算术函数的定义。方法是把函数声明为类 `Ratio` 的友元。

一个友元函数是一个非成员的函数，它可以访问声明的类中的所有成员。因此它有一个成员函数的所有优先权，尽管它实际上不是类的成员函数。这种性质主要用在重载运算符上。

#### 例 11.4 把乘法运算符声明为一个友元

下面是类 `Ratio` 的声明，其中重载的乘法运算符声明为了一个友元函数：

```
class Ratio
{   friend Ratio operator* (const Ratio&, const Ratio&);
public:
    Ratio (int = 0, int = 1);
    Ratio (const Ratio&);
    Ratio& operator= (const Ratio&);
    //其他声明
private:
    int num, den;
    //其他声明
};
```

注意函数原型插入了类声明的内部，在 `public` 部分的上面。也要注意函数的两个参数是由常量参数传递的。

现在可以按预期来实现这个非成员：

```
Ratio operator* (const Ratio& x, const Ratio& y)
{ Ratio z (x.num * y.num, x.den * y.den);
  return z;
}
```

注意关键词 `friend` 没有用到函数实现部分；也要注意到生存空间解析运算的前缀 `Ratio::` 没有用到，因为它不是一个成员函数。

下面是一个用到改进了的类 `Ratio` 的小程序。

**例 11.5 含有赋值和乘法操作的类 Ratio**

```
#include "Ratio.h"
int main ()
{ Ratio x (22, 7), y (-3, 8), z;
  z = x;           // 调用赋值操作
  z.print ();      cout << endl;
  x = y * z;       // 调用乘法操作
  x.print ();      cout << endl;
}
22/7
- 33/28
```

注意在构造函数中调用了 `reduce ( )` 函数来把  $-66/56$  降为  $-33/58$  (见例 10.8)。

## 11.5 算术赋值运算符重载

C++ 允许把算术运算符和赋值运算符组合在一起。例如, 用  $x *= y$  来代替  $x = x * y$ 。在类中应用时, 这些组合运算符都可以用。

**例 11.6 含有运算符  $*=$  的类 Ratio**

```
class Ratio
{
public:
  Ratio (int = 0, int = 1);
  Ratio& operator = (const Ratio&);
  Ratio& operator * = (const Ratio&);
  //其他声明
private:
  int num, den;
  //其他声明
};

Ratio& Ratio::operator * = (const Ratio& r);
{ num = num * r.num;
  den = den * r.den;
  return * this;
}

x *= y  x = z
```

运算符 `operator * =` 和基本的赋值运算符 `operator =` 有相同的语法和几乎一样的实现。例如, 下面的两行有相同的效果, 即使他们调用了不同的运算符:

```
x = x * y;
x *= y
```

## 11.6 关系运算符重载

算术运算符能作为友元函数重载, 六个关系运算符  $<$ 、 $>$ 、 $<=$ 、 $>=$ 、 $==$  和  $!=$  能以

相同的方式重载。

### 例 11.7 类 Ratio 中 == 运算符的重载

```
class Ratio
{
    friend bool operator == (const Ratio&, const Ratio&);
    friend Ratio operator * (const Ratio&, const Ratio&);
    //其他声明
public:
    Ratio (int = 0, int = 1);
    Ratio (const Ratio&);
    Ratio& operator = (const Ratio&);
    //其他声明
private:
    int num, den;
    //其他声明
};

bool operator == (const Ratio& x, const Ratio& y)
{
    return (x.num * y.den == y.num * x.den);
}
```

两个分数  $a/b$  和  $c/d$  相等这个测试等价于测试  $a * d = b * c$ 。因此把等于运算符用于整型数以此来为类 Ratio 定义等于运算符并以此结束。

注意关系运算符返回一个 int 类型，代表着或者 “true” (1) 或者 “false” (0)。

## 11.7 字符串运算符重载

C++ 允许重载流插入运算符 >> 是为了用户的输入，允许重载流取出运算符 << 是为了用户的输出。如算术和关系运算符，这些也可以声明为友元函数。

对一个类 T，它有数据成员 d，输出运算符的语法是：

```
friend ostream& operator<< (ostream& ostr, const T& t)
{ return ostr << t.d; }
```

这里的 ostream 是头文件 iostream.h 中直接定义的一个标准类。注意，所有的参数和返回值都是通过引用来传递的。

这个函数可以用与基础类型一样的方式调用：

```
cout << "x = " << x << ", y = " << y << endl;
```

下面是一个如何写习惯的输出的例子。

### 例 11.8 对类 Ratio 重载输出运算符

```
class Ratio
{
    friend ostream& operator<< (ostream&, const Ratio&);
public:
    Ratio (int n=0, int d=1) : num (n), den (d) {}
}
```

```

        //其他声明
    private:
        int num, den;
        //其他声明
};

int main ()
{
    Ratio x (22, 7), y (-3, 8);
    cout << "x = " << x << ", y = " << y << endl;
}

ostream& operator<< (ostream& ostr, const Ratio& r)
{
    return ostr << r.num << '/' << r.den;
}

```

$x = 22/7, y = -3/8$

当 `main ()` 函数第二行执行时，表达式 `cout << "x = "` 首先执行。它调用标准输出运算符 `<<`，把标准输出流 `cout` 和字符串 `"x = "` 传递给它。如平常一样，它把字符串插入到输出流中，然后返回一个引用给 `cout`。然后这个返回值随着对象 `x` 传递给重载的 `<<` 运算符。这次对 `operator` 的调用执行时用 `cout` 代替 `ostr`，用 `x` 代替 `r`。结果是执行下面一行。

```
return ostr << r.num << '/' << r.den;
```

它把 `22/7` 插入了输出流，返回一个对 `cout` 的引用。然后是又一次对标准输出运算符 `<<` 的调用以及又一次对重载运算符的调用，每次的调用结果（对 `cout` 的引用）作为对下次调用的输出级联起来。最后，对标准结果运算符 `<<` 作最后一次调用，把 `cout` 和 `endl` 传递过来。这样促进流的流动，结果是打印下面这行：

$x = 22/7, y = -3/8$

类 `T` 的关于数据成员 `d` 的重载输入运算符的语法如下。

```
friend istream& operator>> (istream& istr, T& t)
{ return istr >> t.d; }
```

这里的 `istream` 是头文件 `istream.h` 中间接定义的另一标准类。

下面是一个关于如何写用户输入的例子。

### 例 11.9 在类 `Ratio` 中重载输入运算符

```

class Ratio
{
    friend istream& operator>> (istream&, Ratio&);
    friend ostream& operator<< (ostream&, const Ratio&);
public:
    Ratio (int n=0, int d=1) : num (n), den (d) {}
    //其他声明
private:
    int num, den;
    int gcd (int, int);
    void reduce ();
};

int main ()
{
    Ratio x, y;
}

```

```

cin >> x > y;
cout << "x = " << x << ", y = " << y << endl;

istream& operator>> (istream& ist1, Ratio& r)
{
    cout << "\t Numerator: "; ist1 >> r.num;
    cout << "\t Denominator: "; ist1 >> r.den;
    r.reduce();
    return ist1;
}

```

```

        Numerator: -10
        Denominator: -24
        Numerator: 36
        Denominator: -20
x = 5/12, y = -9/5

```

输入运算符的这个版本包括了便于输入的用户提示。它还包含了一个对效用函数 `reduce()` 的调用。注意, 作为一个友元, 运算符可以访问私有型函数。

## 11.8 运算符转换

在类 `Ratio` (例 10.1) 的最初实现中定义了成员函数 `convert()`, 以此把 `Ratio` 类型转换成 `double` 类型:

```
double convert() { return double(num)/den; }
```

它要求成员函数作如下调用:

```
x.convert();
```

我们有个目标: 那就是使类 `Ratio` 的对象与基本类型的对象 (也就是说变通的变量) 表现相像, 为了与目标保持一致, 将有一个转换函数, 它可以用与普通的类型转换一致的语法调用:

```

n = int(t);
v = double(x);

```

这可以通过一个转换运算符来完成。

类 `Ratio` 已有了把对象从 `int` 到 `Ratio` 转换的设备:

```
Ratio x(22);
```

这可以用默认的构造函数来处理, 它把 22 赋给了 `x.num`, 把 1 赋给了 `x.den`。这个构造函数也能处理直接的从 `int` 到 `Ratio` 的类型转换:

```
x = Ratio(22);
```

一个给定类的构造函数用来把另一种类型转换成那个类的类型

把给定的类的类型转换成另一种类型需要一个不同种类的成员函数。这就是转换运算符, 它有着不同的语法。如果 `type` 是对象要转换成的类型, 那么转换运算符可以如下声明:

```
operator type ( );
```

例如，返回一个等价的 float 的类 Ratio 的一个成员函数将会如下声明：

```
operator float ( );
```

或者，如果想让它转换成 double 型，那么将做如下声明：

```
operator double ( );
```

并且，如果想让它对常数 Ratio 型的对象（例如 pi）也是可用的，那么将做如下声明：

```
operator double ( ) const;
```

在类 Ratio 的最初实现（例 10.1）中已为此目的定义了成员函数 convert（）。

### 例 11.10 为类 Ratio 增加一个转换运算符

```
class Ratio
{
    friend ostream& operator >> (ostream&, Ratio&);
    friend ostream& operator << (ostream&, const Ratio&);
public:
    Ratio (int n=0, int d=1) : num (n), den (d) {}
    operator double ( ) const;
private:
    int num, den;
};

int main ()
{
    Ratio x (-5, 8);
    cout << "x = " << x << ", double (x) = " << double (x) << endl;
    const Ratio p (22, 7);
    const double PI = double (p);
    cout << "p = " << p << ", PI = " << PI << endl;
}

Ratio: operator double ( ) const
{ return double (num) /den;
}

x = -5/8, double (x) = -0.625
p = 22/7, PI = 3.14286
```

首先，用转换运算符 double（）把 Ratio 型对象转换成 double 型的 -0.625。然后再用它把常量 Ratio 型对象 p 转换成常数 double 型的 pi。

## 11.9 加减运算符重载

每个 ++ 运算符和 -- 运算符都有两种形式：前缀和后缀。这四种形式的每一种都可以重载。下面将考查加号运算符的重载，减号运算符的重载类似。

当应用于整数时，前加运算符只是把正被增加的对象值加 1，它是个一元运算符。它的惟一参数就是正被增加的对象。对名为 T 的类重载前加运算符的句法简单表述为：

```
T operator ++ ();
```

对 Ratio 类，它如下声明：

```
Ratio operator ++ ();
```

### 例 11.11 为类 Ratio 加一个前加运算符

这个例子为 Ratio 类增加一个前加运算符，尽管可以让这个函数做我们需要的任何事情，它还是应该和标准前加运算符对整型的作用相一致。在对象的值被用在表达式中之前，它把对象的当前值加 1。这等价于把它的分母加到它的分子上：

$$\frac{22}{7} + 1 = \frac{22+7}{7} = \frac{29}{7}$$

因此，只是把 den 加到 num，然后返回 \* this，它是对象本身。

```
Class Ratio
{
    friend ostream& operator<< (ostream&, const Ratio&);
public:
    Ratio (int n=0, int d=1) : num (n), den (d) {}
    Ratio operator ++ ();
    // 其他声明
private:
    int num, den;
    // 其他声明
};

int main ()
{
    Ratio x (22, 7), y = ++x;
    cout << "y = " << y << ", x = " << x << endl;

    Ratio Ratio:: operator ++ ()
    {
        num += den;
        return *this;
    }

    y = 29/7, x = 29/7
}
```

后缀运算符和前缀运算符有相同的功能。例如，前加运算符和后加运算符都命名为 operator ++。为了区别它们，C++ 指定了前缀运算符有一个参数，后缀有两个参数（应用时，它们都表现为带一个参数）。因此，一个重载的后加运算符的原型的正确语法是：

```
T operator ++ (int);
```

所要求的参数必须是 int 型的。这看起来有点奇怪，因为当它被调用时，并没有整数传给函数。因此，这个整数参数是一个哑元参数，这样要求可以把后缀运算符从相应的前缀运算符中区别出来。

### 例 11.12 为类 Ratio 增加一个后加运算符

为了与通常的关于整型的后加运算符相一致，这种重载的版本应该在 x 的值赋给 y 以后再改变它。为了做到这一点，需要一个临时的对象来保持属于这个调用的对象的内容。这可



以通过把 \* this 赋给临时对象来完成。然后在把 den 加到 num 后这个对象可以返回。

```

class Ratio
{
    friend ostream& operator << (ostream&, const Ratio&);
public:
    Ratio (int n=0, int d=1) : num (n), den (d) {}
    Ratio operator++ ();          //前加
    Ratio operator++ (int);       //后加
private:
    int num, den;
};

int main ()
{
    Ratio x (22, 7), y = x++;
    cout << "y = " << y << ", x = " << x << endl;
}

Ratio Ratio::operator++ (int)
{
    Ratio temp = *this;
    num += den;
    return temp;
}

```

y = 22/7, x = 29/7

注意，在 operator++ 函数中哑元参数是一个未命名的 int。由于未用到它，所以不必要命名。但它必须声明以此来区分后加运算符和前加运算符。

## 11.10 下标运算符重载

回忆一下，如果 a 是一个数组，那么表达式 a[i] 的真正意义是 \* (a+i)。这是因为 a 实际上是数组的初始元素的地址，由于对 a 增加的字节数是每个数组元素大小的 i 倍，因此 a+i 是第 i 个元素的地址。

符号 [ ] 表示下标运算符。它的名字起源于数组的初始应用，此时 a[i] 表示数学符号  $a_i$ 。当用做 a[i] 时，它有两个操作数 a 和 i。表达式 a[i] 等价于 operator [] (a, i)。作为运算符，[ ] 可以重载。

### 例 11.13 对类 Ratio 增加一个下标运算符

```

class Ratio
{
    friend ostream& operator << (ostream&, const Ratio&);
public:
    Ratio (int n=0, int d=1) : num (n), den (d) {}
    int& operator [] (int);
    //其他声明
private:
    int num, den;
    //其他声明
}

```

```

;

int main ()
{ Ratio x (22, 7);
  cout << "x = " << x << endl;
  cout << "x [1] = " << x [1] << ", x [2] = " << x [2] << endl;
;

ostream& operator<< (ostream& ostr, const Ratio& r)
{ return ostr << r.num << "/" << r.den;

int Ratio::operator [] (int i)
{ if (i == 1) return num;
  else return den;

  x = 22/7
  x [1] = 22, x [2] = 7

```

表达式 `x [1]` 调用下标运算符, 把 1 传给 `i`, 返回的是 `x.num`。类似地, `x [2]` 返回的是 `x.den`。如果 `i` 有不是 1 或 2 的其他值, 那么标准错误流 `cerr` 收到一个错误的信息, 然后调用 `exit ()` 函数。

这个例子是人造的。用 `x [1]` 和 `x [2]` 而不是 `x.num` 和 `x.den` 去访问 `Ratio` 类对象 `x` 的区域并没什么好处。然而在很多重要的类中下标十分有用。(见习题 11.2)

注意下标运算符是一个访问函数, 由于它提供了公有型对私有成员数据的访问。

## 复 习 题

- 11.1 关键词 `operator` 如何用?
- 11.2 `* this` 经常指向什么?
- 11.3 为什么 `this` 指针不能用于非成员函数?
- 11.4 为什么重载的赋值运算符返回 `* this`?
- 11.5 下面两个声明的结果有什么不同:

```

Ratio y (x);
Ratio y = x;

```

- 11.6 下面两行语句的结果有什么不同:

```

Ratio y = x;
Ratio y; y = x;

```

- 11.7 为什么 `**` 不能重载为求幂运算符?
- 11.8 为什么流运算符 `<<` 和 `>>` 应该重载为友元函数?
- 11.9 为什么算术运算符 `+`、`-`、`*`、`/` 应该重载为友元函数?
- 11.10 重载的前加运算符定义与后加运算符定义如何区别?

- 11.11 为什么在后加运算符的实现中 int 参数没有命名?
- 11.12 例如: `v[2] = 22`, 是什么机理允许重载的下标运算符可以用在赋值语句的左边?

## 习 题

- 11.1 为类 `Ratio` 实现二进制的减法运算符、一元的负运算符和小于运算符 (见例 11.4)。
- 11.2 实现一个 `Vector` 类, 带有一个默认的构造函数、一个复制构造函数、一个析构函数、重载的赋值运算符、下标运算符、等于运算符、流插入运算符和流取出运算符。
- 11.3 为类 `Ratio` 实现加法和除法操作符 (见例 11.5)。
- 11.4 为类 `Ratio` 重写重载的输入运算符, 使它读入一个如 “22/7” 的分数, 而不是为分母和分子加提示。
- 11.5 为类 `Point` 实现一个重载的赋值运算符 (见习题 10.1)。
- 11.6 为类 `Point` 实现重载的流插入运算符 (见习题 10.1)。
- 11.7 为类 `Point` 实现重载的比较关系运算符 `==` 和 `!=` (见习题 10.1)。
- 11.8 为类 `Point` 实现重载的加法 `+` 和减法运算符 `-` (见习题 10.1)。
- 11.9 实现一个重载的乘法运算符 `*` 来返回两个 `Point` 类对象的点积 (见习题 10.1)。

## 复习题答案

- 11.1 关键词 `operator` 是用来格式化重载一个运算符的函数的名字。例如, 重载赋值运算符 `=` 的函数的名字是 “`operator=`”。
- 11.2 表达式 `* this` 总是指向拥有当前表达式所在成员函数的对象, 因此它只能在成员函数中使用。
- 11.3 关键字 `this` 是指向拥有此表达式所在成员函数的对象的指针。
- 11.4 重载的赋值运算符应该返回 `* this`, 这样赋值运算符就能用在连续的调用了, 如:
- ```
w = x = y = z;
```
- 11.5 结果没有差别。两声明都用复制的构造函数来创建对象 `y` 作为对象 `x` 的复制。
- 11.6 声明 `Ratio y = x;` 调用了复制的构造函数。代码 `Ratio y; y = x;` 调用了默认的构造函数和赋值运算符。
- 11.7 由于符号 `**` 不是一个 C++ 运算符, 所以它不能重载为一个运算符。
- 11.8 流运算符 `<<` 和 `>>` 应该被重载为友元函数, 因为它们的左操作数应为流对象。如果一个重载的运算符是一个成员函数, 那么它的左操作数是 `* this`, 它是类的一个对象, 而函数是类的成员。
- 11.9 数学运算符 `+`、`-`、`*`、`/` 应该被重载为友元函数, 这样它们左边的运算数可以被声明为常量, 如 `22 + x` 这样的表达式就被允许了。如果被重载的运算符是成员函数, 则它左边的运算数是 `* this`, 它不是常量。

- 11.10 重载的前加运算符没有参数。重载的后加运算符有一个 int 类型的参数。
- 11.11 在后加运算符的实现中 int 参数没有命名是因为没有用到它，它是一个哑元参数。
- 11.12 通过返回一个引用，重载的下标运算符 [ ] 能用于一个赋值语句的左边，如：  
 $v[2] = 22$ 。这是因为，作为一个引用， $v[2]$  是一个值。

## 习题答案

- 11.1 所有的这三个操作都实现为友元函数，以方便它们对宿主对象的数据成员 num 和 den 的访问：

```
class Ratio
{
    friend Ratio operator~ (const Ratio&, const Ratio&);
    friend Ratio operator - (const Ratio&);
    friend bool operator < (const Ratio&, const Ratio&);
public:
    Ratio (int = 0, int = 1);
    Ratio (const Ratio&);
    Ratio& operator = (const Ratio&);
    //其他声明
private:
    int num, den;
    int gcd (int, int);
    int reduce ();
};
```

二进制减法运算符只是构造和返回了一个 Ratio 对象 z，它表示  $x - y$  的差：

```
Ratio operator - (const Ratio& x, const Ratio& y)
{
    Ratio z (x.num * y.den - y.num * x.den, x.den * y.den);
    z.reduce ();
    return z;
}
```

代数上， $a/b - c/d$  是通常用分母  $bd$  表示的：

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$$

因此， $x - y$  的分子应该是  $x.num * y.den - y.num * x.den$ ，分母应该是  $x.den * y.den$ 。函数用这个分母和分子构造了 Ratio 型的对象 z。无论 x，y 是多少，代数公式能产生一个不是简化形式的分数。例如， $1/2 - 1/6 = (1 \cdot 6 - 2 \cdot 1) / (2 \cdot 6) = 4/12$ 。因此，在返回结果对象 z 之前调用了 reduce 效用函数。

一元负运算符重载了符号“-”。它可以根据它的参数表和二进制减法运算符区分开来；它只有一个参数：

```
Ratio Ratio::operator~ (const Ratio& x)
{
    Ratio y (-x.num, x.den);
    return y;
}
```

为了否定分数  $a/b$ , 只要否定它的分子就行了。因此, 新构造的 Ratio 对象  $y$  和  $x$  有相同的分母, 但它的分子是  $-x.num$ 。如果更改一下默认的构造函数以确保每个对象的  $den$  值是正的话, 小于运算符就比较容易实现了。然后, 可以把标准的等价式用于小于运算符:

$$\frac{a}{b} < \frac{c}{d} \Leftrightarrow ad < bc$$

```
bool operator< (const Ratio& x, const Ratio& y);
|   return (x.num * y.den < y.num * x.den);
|
|
Ratio::Ratio (int n, int d) : num (n), den (d)
|   if (d == 0) n = 0;
|   else if (d < 0) { n * = -1; d * = -1; }
|   reduce ();
```

由于效用应该被每个成员函数调用来更改  $den$ , 更改可以确保在 `reduce()` 函数中  $den > 0$  能够实现。然而, 其他任何一个成员函数都不允许  $den$  符号变化, 因此, 当对象创建时, 通过把它规定为正的, 就不必再检查条件了。

## 11.2 下面是这个类的声明:

```
class Vector
|   friend bool operator== (const Vector&, const Vector&);
|   friend ostream& operator<< (ostream&, const Vector&);
|   friend istream& operator>> (istream&, Vector&);
public:
|   Vector (int i, double = 0, 0);           //默认的构造函数
|   Vector (const Vector&);                 //复制的构造函数
|   ~Vector ();                             //析构函数
|   const Vector& operator= (const Vector&); //赋值运算符
|   double& operator [] (int) const;        //下标运算符
private:
|   int size;
|   double* data;
|;
```

下面是重载的等于关系运算符的实现:

```
bool operator== (const Vector& v, const Vector& w)
|   if (v.size != w.size) return 0;
|   for (int i = 0; i < v.size; i++)
|       if (v.data[i] != w.data[i]) return 0;
|   return 1;
|
```

这是一个非成员函数, 它根据两个向量  $v$ ,  $w$  是否相等而返回 1 或 0。如果它们的  $size$  不等, 那么它立即返回 0。否则它检查两个向量的相应成员, 每次检查一个。如果不

一致, 那么它又立即返回 0。只有当全部的循环完成而没有发现任何的不一致时, 才能得出两个向量相等的结论, 并且返回 1。

下面是重载的流插入运算符的实现:

```
ostream& operator<< (ostream& ostr, const Vector& v)
{ ostr<<" (" ;
  int i;
  for (int i=0; i<v.size(); i++)
    ostr<<" ["<i>i</i>"<<" ";
    if ((i+1)%8 == 0) cout<<" \n ";
  }
  return ostr<<" v ["<i>i</i>"<<" )";
}
```

程序打印向量如下: [1.11111, 2.22222, 3.33333, 4.44444, 5.55555]。如果向量的元素超过 8 个的话, 循环内部的条件会使输出绕成整洁的几行。

输出传给 ostr, 它只是传递给函数的输出流的一个局部名字。如果函数如下调用: cout<<v; 那么就会是 cout 了。在函数最后几行, 表达式 ostr<<v[i]<<" )\n"; 两次调用 (标准的) 流取出运算符。这两次调用把 ostr 作为表达式的值返回, 然后对象 ostr 被函数返回。

下面是重载的流取出运算符:

```
istream& operator>> (istream& istr, Vector& v)
{ for (int i=0; i<v.size(); i++)
  { cout<<" ["<i>i</i>"<<" ";
    istr>>v[i];
  }
  return istr;
}
```

这个实现为向量 v 的每个元素提示用户。它也可以不要用户提示来实现, 只须每次读一个元素。注意元素是从 istr 流中读入的, 它是传给这个函数的第一个参数。当函数被象 cin>>v; 这样实现时, 标准输入流 cin 被传给参数 istr, 这样向量的成员就确实被从 cin 中读入了。参数 istr 只是实际输入流, 可能是 cin, 的本地名称。注意这个参数也被返回, 允许如 cin>>u>>v>>w, 这样的串联使用。

这里是缺省构造函数的实现

```
Vector::Vector (int sz, double t): size (sz)
{ data = new double [size];
  for (int i=0; i<size; i++)
    data [i] = t;
}
```

声明 Vector u; 将构造有一个成员的向量 u, 它的值是 0.0。声明 Vector v (4) 将构造有 4 个成员的向量 v, 它们的值都为 0.0。声明 Vector w (8, 3.14159) 将构造有 8 个成员的向量 w, 它们的值都是 3.14159。

这个构造函数使用了初始化列表 size (sz) 来将参数 sz 赋给数据成员 size。接着它使

用 new 运算符分配这么多元素的内存给数组 data。最后，它将所有的元素赋值 1。拷贝构造函数与缺省构造函数基本差不多：

```
Vector::Vector (const Vector& v) : size (v.size)
{
    data = new double [v.size];
    for (int i = 0; i < size; i++)
        data [i] = v.data [i];
}
```

它使用参数向量 v 的数据成员对被构造的对象运行初始化。因此它将 v.size 赋给新对象的 size 成员，并将 v.data [i] 赋给新对象的 data 成员。

析构函数只是简单释放分配给 data 数组的内存，将 data 设为 NULL，并将 size 设为 0：

```
Vector::~~Vector ()
{
    delete [] data;
    data = NULL;
    size = 0;
}
```

被重载的赋值运算符创建了一个向量 v 的复制品：

```
const Vector& Vector::operator = (const Vector& v)
{
    if (&v != this)
    {
        delete [] data;
        size = v.size;
        data = new double [v.size];
        for (int i = 0; i < size; i++)
            data [i] = v.data [i];
    }
    return *this;
}
```

条件 (&v != this) 判断进行调用的对象是否与向量 v 相同。如果 v 的地址与 this 相同 (也就是当前对象的地址)，则它们是相同的对象，没有必要进行操作。这种检查是一种安全上的预防，防止一个对象直接或间接地被赋给自己，如：w = v = w；

在创建新对象前，函数释放分配的数据数组，接着用与拷贝构造函数相同的方法复制向量 v。

被重载的角注运算符只是简单地返回对象 data 数组的第 i 个成员。

```
double& Vector::operator [] (int i) const
{
    return data [i];
}
```

```
11.3 Ratio operator+ (const Ratio& r1, const Ratio& r2)
{
    Ratio r (r1.num * r2.den + r2.num * r1.den, r1.den * r2.den)
    r.reduce ();
    return r;
}

Ratio operator/ (const Ratio& r1, const Ratio& r2)
{
    Ratio r (r1.num * r2.den, r1.den * r2.num);
    r.reduce ();
    return r;
}
```

- ```

11.4 ostream& operator<< (ostream& ostr, const Ratio& r)
    | return ostr << r.num << "/" << r.den;
    |

11.5 Point& Point:: operator= (const Point& point)
    | _x = point._x;
    | _y = point._y;
    | _z = point._z;
    | return *this;
    |

11.6 ostream& operator<< (ostream& ostr, const point& point)
    | return ostr << " (" << _x << ", " << _y << ", " << _z << ")";
    |

11.7 bool Point:: operator== (const Point& point) const
    | return _x == point._x && _y == point._y && _z == point._z;
    |
    bool Point:: operator!= (const Point& point) const
    | return _x != point._x || _y != point._y || _z != point._z;
    |

11.8 Point operator+ (const Point& p1, const Point& p2)
    | return Point (p1._x+p2._x, p1._y+p2._y, p1._z+p2._z);
    |
    Point operator- (const Point& p1, const Point& p2)
    | return Point (p1._x-p2._x, p1._y-p2._y, p1._z-p2._z);
    |

11.9 Point operator* (const double coef, const Point& point)
    | return Point (coef*point._x, coef*point._y, coef*point._z);
    |

```



## 第 12 章 组合和继承

### 12.1 引言

经常需要用存在的类来定义新的类。实现它的两个方法叫做组合和继承。这一章就描述这两个方法，并指出如何决定何时应用它们。

### 12.2 组合

类的组合（也叫包含或聚合）指的是在另一个类的定义中用到了一个或更多的类。当新类的数据成员是另一个类的对象时，称这个新类是其他对象的合成物。

#### 例 12.1 一个 Person 类

下面是一个表示人的类的简单定义。

```
class Person
{
public:
    Person (char * n="", char * nat="U.S.A.", int s=1)
        : name (n), nationality (nat), sex (s) {}
    void printName () { cout << name; }
    void printNationality () { cout << nationality; }
private:
    string name, nationality;
    int sex;
};
```

```
int main ()
{
    Person creator ("Bjame Stroustrup", "Denmark");
    cout << "The creator of C++ was ";
    creator.printName ();
    cout << ", who was born in ";
    creator.printNationality ();
    cout << ".\n";
}
```

The creator of C++ was Bjame Stroustrup, who was born in Denmark.

这个例子说明了在 Person 类内 string 类的组合。下一个例子定义了另一个类，能用 Person 类来构造改善它。

## 例 12.2 一个 Date 类

```

class Date
{
    friend ostream& operator >> (ostream&, Date&);
    friend ostream& operator << (ostream&, const Date&);
public:
    Date ( int m=0, int d=0, int y=0) : month (m), day (d), year (y) {}
    void setDate (int m, int d, int y) : month = m; day = d; year = y;
private:
    int month, day, year;
};

ostream& operator >> (ostream& out, Date& x)
{
    int >> x.month >> x.day >> x.year;
    return out;
}

ostream& operator << (ostream& out, const Date& x)
{
    static char * monthName [13] = {"", "January", "February",
        "March", "April", "May", "June", "July", "August",
        "September", "October", "November", "December"};
    out << monthName [x.month] << " " << x.day << " " << x.year;
    return out;
}

int main ()
{
    Date peace (11, 11, 1918);
    cout << "World War I ended on " << peace << ".\n";
    peace.setDate (8, 14, 1945);
    cout << "World War II ended on " << peace << ".\n";
    cout << "Enter month, day, and year: ";
    Date date;
    cin >> date;
    cout << "The date is " << date << ".\n";
}

World War I ended on November on November 11, 1918.
World War II ended on August 14, 1945.
Enter month, day, and year: 7 4 1776
The date is July 4, 1776.

```

测试驱动程序测试出默认的构造函数 setDate () 函数，重载的插入运算符 <<，重载的取出运算符 >>

现在可以用 Person 类中的 Date 类来存储人的生日和死亡日期

## 例 12.3 在 Person 中组合 Date 类

```
#include "Date.h"
```

```

class Person
{
public:

```

```

    Person (const n="", int s=0, char * nat="U.S.A.")
        : name (n), sex (s), nationality (nat) {}
    void setDOB (int m, int d, int y) {dob.setDate (m, d, y);}
    void setDOD (int m, int d, int y) {dod.setDate (m, d, y);}
    void printName () {cout << name;}
    void printNationality () {cout << nationality;}
    void printDOP () {cout << dob;}
    void printDOD () {cout << dod;}
private:
    string name, nationality;
    Date dob, dod;           // 出生日期, 死亡日期
    int sex;                  // 0 = female, 1 = male
};

int main ()
{
    Person author ("Thomas Jefferson", 1);
    author.setDOB (4, 13, 1743);
    author.setDOD (7, 4, 1826);
    cout << "The author of the Declaration of Independence was ";
    author.printName ();
    cout << ". He was born on ";
    author.printDOB ();
    cout << " and he died on ";
    author.printDOD ();
    cout << ".\n";
}

```

The author of the Declaration of Independence was Thomas Jefferson.  
He was born on April 13, 1743 and died on July 4, 1826.

再一次注意, 已用一个类的成员函数来定义被组合类的成员函数: setDate () 函数是用来定义 setDob () 函数和 setDod () 函数的。

组合通常是指一个“有一个”的关系, 因为组合类的对象“有”被构成类的对象作为它的成员。Person 类的每个对象“有一个”string 类型的 name 对象及“有一个”string 类型的 nationality 对象。组合是一种重用已有的软件来创建新软件的方法。

## 12.3 继承

另一种重用存在的软件来创建新的软件的方法是继承 (也叫做适应或派生)。它通常是指一个“是一个”的关系, 因为这个类的每个所定义的对象也“是”一个被继承类的对象。

类 Y 继承 X 的一般语法是:

```

class Y : public X {
// ...
};

```

这里, X 被称做基类 (或超类), Y 被称做派生类 (或子类)。冒号后的关键字 public 描述了 public inheritance (公共继承), 它的意思是继类中的公共成员在派生类中也成为公共成员。

### 例 12.4 从 Person 中派生 Student 类

```
#include "Person.h"

class Student : public Person
{ public:
    Student (char* n, int s=0, char* i="")
        : Person (n, s), id (i), credits (0) {}
    void setDOM (int m, int d, int y) { dom.setDate (n, d, y); }
    void printDOM () { cout << dom; }
private:
    string id;           // 学生的识别号码
    Date dom;           // 入学考试日期
    int credits;         // 课程学分
    float gpa;          // 年级平均分
};
```

Student 类继承了 Person 类的所有 public 功能，包括了 Person () 构造函数，Person 类用该函数初始化 Person 类中的 name。值得注意的是，它是 Person 类的一个 private (私有) 成员，因此它不能直接访问。

下面是 Student 类的一个测试驱动程序：

```
#include "Student.h"
int main ()
{ Student x ("Ann Jones", 0, "219360061");
  x.setDOB (5, 13, 1997);
  x.setDOM (8, 29, 1995);
  x.printName ();
  cout << "\n\t\t\t\t\tBorn: "; x.printDOB ();
  cout << "\n\t\t\t\t\tMatriculated: "; x.printDOM (); cout << endl;
```

Ann Jones

Born: May 13, 1997

Matriculated: August 29, 1995

## 12.4 保护型类成员

在 12.3 节中的 Student 类有一个重要的问题：它不能直接访问它的 Person 超类中的私有型的数据成员：name、nationality、DOB、DOD 和 sex。不能访问前四个私有数据成员并不是很严重的问题，因为可以通过 Person 类的构造函数和公共访问函数进行读写，但是无法对 Student 中的 sex 进行读或写。解决这个问题的一个方法是使 sex 成为 Student 类的一个数据成员。但这样很不自然，因为 sex 是所有 Person 型对象所拥有的属性，而不仅限于 Student 型对象。一个解决方案是，在 Person 类中把 private 访问标识符变为 protected。它允许从派生中对这些数据成员进行访问。

### 例 12.5 带有保护型数据成员的 Person 类

这和前面所给出的两个例子中的类定义相同，只是 private 访问标识符已变成 protected，同时在 Student 类中加入了访问函数 printSex ()：

```
#include "Date.h"

class Person
{
public:
    Person (char * n = "", int s = 0, char * nat = "U.S.A.")
        : name (n), sex (s), nationality (nat) {}
    void setDOB (int m, int d, int y) { dob.setDate (m, d, y); }
    void setDOD (int m, int d, int y) { dod.setDate (m, d, y); }
    void printName () { cout << name; }
    void printNationality () { cout << nationality; }
    void printDOB () { cout << dob; }
    void printDOD () { cout << dod; }
protected:
    string name, nationality;
    Date dob, dod;           // 出生、死亡日期
    int sex;                 // 0 = 女, 1 = 男
};

class Student : public Person
{
public:
    Student (char * n, int s = 0, char * i = "")
        : Person (n, s), id (i), credits (0) {}
    void setDOM (int m, int d, int y) { dom.setDate (m, d, y); }
    void printDOM () { cout << dom; }
    void printSex () { cout << (sex ? "male" : "female"); }
private:
    string id;               // 学生的识别号码
    Date dom;               // 入学考试日期
    int credits;            // 课程学分
    float gpa;              // 年级平均分
};
```

现在 Person 类中定义的几个数据成员都可以被它的 Student 子类访问，如下面的测试驱动程序所示：

```
int main ()
{
    Student x ("Ann Jones", 0, "219360061");
    x.setDOB (5, 13, 1997);
    x.setDOM (8, 29, 1995);
    x.setDOD (7, 4, 1826);
    x.printName ();
    cout << "\n\t Born: " x.printDOB ();
    cout << "\n\t Sex: " x.printSex ();
    cout << "\n\t tMatriculated: " x.printDOM ();
    cout << endl;
}
```

Ann Jones

Born: May 13, 1997

Sex: female

Matriculated: August 29, 1995

Protected 访问类型介于 private 和 public 类型之间, 即私有成员只能从类自身的内部或被它的友元类访问; 保护成员可以从类的内部、它的友元类、它的派生类及派生类的友元类访问; 公共成员可以从文件的任何地方访问。一般来说, 只有当为类定义一个子类时才用保护型代替私有型。

子类继承了它的基类的所有公共和保护型成员。这意味着, 从子类的角度来看, 它基类的公共和保护型成员就像在子类中实际声明的一样。例如, 假设类 X 和子类 Y 如下定义:

```
class X
{
public:
    int a;
protected:
    int b;
private:
    int c;
};

class Y : public X
{
public:
    int d;
};
```

并且 x 和 y 如下声明:

```
X x;
Y y;
```

则可以把对象 x, y 形象化如图 12.1 所示:

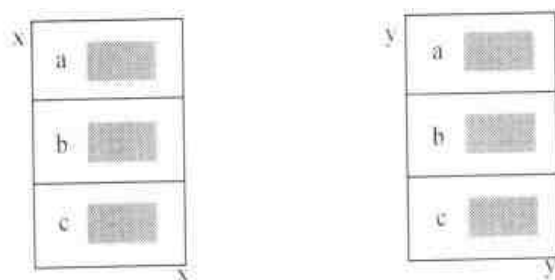


图 12.1 对象 x 和 y 示例

类 X 的公有型成员 a 被继承为 y 的一个公有型成员, 类 X 的保护型成员 b 被继承为 y 的一个保护型成员, 但 X 的私有型成员 c 不能被 y 继承 (每个对象中的水平线表明了对象的公共型、保护型和私有型区域)

## 12.5 重载和操纵继承成员

如果  $Y$  是  $X$  的子类, 那么  $Y$  的对象继承了  $X$  的所有公有型和保护型的数据成员和成员函数。例如, `Person` 类中的 `name` 成员和 `PrintName` 函数也是类 `Student` 的成员。

在某些情况下, 可能需要定义一个被继承成员的局部模式。例如, 如果  $a$  是  $X$  的数据成员,  $Y$  是  $X$  的一个子类, 那么也可以为  $Y$  定义一个独立的数据成员  $a$ 。在这种情况下, 称在  $Y$  中定义的  $a$  操纵在  $X$  中定义的  $a$ 。然后类  $Y$  的一个对象  $y$  的引用  $y.a$  将访问在  $Y$  中定义的  $a$  而不是在  $X$  中定义的  $a$ 。为了访问在  $X$  中定义的  $a$ , 可以用  $y.X::a$ 。

同样的规则也适用于成员函数。如果函数  $f()$  在  $X$  中定义, 并且具有同样符号的函数  $f()$  在  $Y$  中定义, 那么  $y.f()$  调用后一个函数, 而  $y.X::a$  调用前一个函数。在这种情况下, 局部函数  $y.f()$  重载了  $X$  中定义的函数  $f()$ , 除非它被调用为  $y.X::a$ 。

这些函数在下面的例子中可以显示

### 例 12.6 操纵一个数据成员, 重载一个成员函数

下面是两个类  $X$  和  $Y$ ,  $Y$  继承了  $X$ 。

```
class X
{
public:
    void f () { cout << "X:: f () executing\n"; }
    int a;
};

class Y : public X
{
public:
    void f () { cout << "Y:: f () executing\n"; } // 重载 X:: f ()
    int a; // 操纵 X:: a
};
```

但  $Y$  的成员和  $X$  中的成员有相同的名称。因此  $Y$  的成员  $f()$  重载了在  $X$  中定义的  $f()$ ,  $Y$  的数据成员  $a$  操纵  $X$  中定义的  $a$ 。

下面是对这两个类的一个测试驱动程序。

```
int main ()
{
    X x;
    x.a = 22;
    x.f ();
    cout << "x.a = " << x.a << endl;

    Y y;
    y.a = 44; // 把 44 赋给定义在 Y 中的 a
    y.X::a = 66; // 把 66 赋给定义在 X 中的 a
    y.f (); // 调用定义在 Y 中的 f ()
    y.X::f (); // 调用定义在 X 中的 f ()
    cout << "y.a = " << y.a << endl;
    cout << "y.X::a = " << y.X::a << endl;
}
```

```

Xz = y;
cout << "z.a = " << z.a << endl;
.
X::f()   executing
x.a = 22
X::f()   executing
X::f()   executing
y.a = 44
z.a = 66

```

在这里，`y` 访问了两个名字为 `a` 的不同的数据成员和两个名字为 `f()` 的不同的成员函数。默认值是派生类 `Y` 中定义的。生存空间解析运算符以 `X::` 的形式重载默认值，访问在父类 `X` 中定义的相应成员。当 `X` 的对象 `z` 被初始化为 `y` 后，它的 `X` 成员中是这么使用的：`z.a` 被赋值为 `y.X::a`。

图 12.2 表示了一个对象 `x`、`y` 和 `z`：

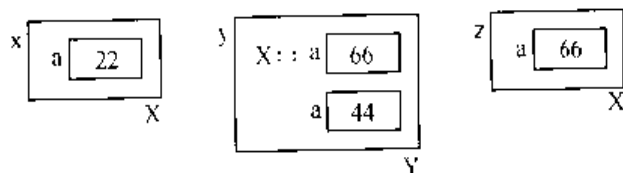


图 12.2 对象 `x`、`y` 和 `z` 示例

例 12.6 和这一章中剩下的大部分例子都是为了表示继承的复杂性而设计的。它们并不是为了例示通常的程序设计练习而有意设计的。相反，它们集中在能适用于更一般情形的 C++ 的独特方面。值得一提的是，例 12.6 所显示的操纵数据成员的方法相当不寻常。尽管重载成员函数并不是不同寻常的，操纵相同类型的数据成员却很少见。更为普通的是对一个不同的类型重复使用相同的数据名，例如：

```

class Y : public X
{
public:
    double a;    //类 X 中的数据成员 a 的类型是整型
}

```

在一个继承的层次结构中，默认的构造函数和析构函数与其他的成员函数作用不同。如下例所示，每个构造函数在执行它之前调用它的父类构造函数，每个析构函数在执行它之后调用它的父类析构函数：

### 例 12.7 父亲构造函数的析构函数

```

class X
{
public:
    X() { cout << "X::X() constructor executing\n"; }
    ~X() { cout << "X::~X() destructor executing\n"; }
};

class Y : public X

```



```

    | public:
      Y () { cout << "Y:: Y () constructor executing\n"; }
      ~Y () { cout << "Y:: Y () destructor executing\n"; }
    };

class Z : public Y
{
    | public:
      Z (int n) { cout << "Z:: Z () constructor executing\n"; }
      ~Z () { cout << "Z:: Z () destructor executing\n"; }
};

int main ()
{
    | Z z (44);
}

```

当 `z` 声明时, 构造函数 `z::z (int)` 被调用。在执行它之前, 它调用构造函数 `Y::Y ()`, 而 `Y::Y ()` 立即调用构造函数 `X::X ()`; 构造函数 `X::X ()` 执行之后, 控制器返回执行完毕的构造函数 `Y::Y ()`。最后是构造函数 `z::z ()` 也执行完毕。结果是父类构造函数以从上到下的顺序执行。

下面是一个更为实用的例子。

### 例 12.8 父亲构造函数和析构造函数

下面是一个演示程序, 它用到一个基类 `Person` 和一个派生类 `Student`:

```

class Person
{
    | public:
      Person (const char * s)
        { name = new char [strlen (s) + 1]; strcpy (name, s); }
      ~Person () { delete [] name; }
    protected:
      char * name;
};

class Student : public Person
{
    | public:
      Student (const char * s, const char * m) : Person (s)
        { major = new char [strlen (m) + 1]; strcpy (major, m); }
      ~Student () { delete [] major; }
    private:
      char * major;
};

int main ()
{
    | Person x ("Bob");
      | Student y ("Sarah", "Biology");
}

```

当 `x` 被初始化的时候, 它调用构造函数 `Person` 分配 4 个字节的内存存储串 “Bob”。然后 `y` 初始化, 首先调用构造函数 `Person` 分配 6 个字节的内存存储串 “Sarah”, 然后分配 8 个

字节的内存存储串“Biology”。y 的范围结束在 x 的范围之前，因为它声明在一个内部块之内

此时，y 的析构函数释放分配给“Biology”的 8 个字节，然后调用析构函数 Person 释放“Sarah”用的 6 个字节。最后，调用析构函数 Person 摧毁 x，释放“Bob”用的 4 个字节。

## 12.6 私有访问对保护访问

私有型和保护型类成员的区别在于，子类能访问父类的保护型成员，但不能访问父类私有型成员。由于保护型更加灵活些，那么什么时候才会把成员变成私有型的呢？答案就在信息隐蔽规则的核心，即当前的限制访问有利于以后的修改。如果将来可能想修改一个数据成员的实现部分，那么把它声明为私有型的将会避免在子类中做必然结果的变化需要。子类对私有型的数据成员是独立的。

### 例 12.9 用保护型和私有型数据成员的 Person 类

假设需要知道一组人员（也就是 Person 对象）是否是高中毕业。只需加入一个保护型数据成员，如 Sex，它存储 0 或 1。但或许以后想用包含关于人的教育程序的更详细信息的成员替代它。因此，现在建立了一个私有型数据成员 hs 来禁止派生类对它的直接访问：

```
class Person
{
public:
    Person (char * n="", int s=0, char * nat="U.S.A.")
        : name (n), sex (s), nationality (nat) {}
    // ...
protected:
    string name, nationality;
    Date dob, dod;           // 出生、死亡日期
    int sex;                  // 0 = 男, 1 = 女
    void setHSgraduate (int g) { hs = g; }
    int isHSgraduate () { return hs; }
private:
    int hs;                   // = 1, 如果是高中毕业
};
```

程序包含了保护型访问函数允许子类访问它的信息。如果以后确实要用其他的数据成员来替代数据成员 hs，则只需更改这两个访问函数的实现部分，这样不会影响其任何子类。

## 12.7 虚函数和多态性

C++ 最有力的特征之一是：它允许不同类型的对象对相同函数的调用反应也不同。这叫做多态性，它是通过虚函数的方式实现的。多态性可能产生是基于这个事实：一个指向基类实例的指针也可以指向其任何子类的实例。

```

class X
{
    // ...

class Y : public X
{
    // ...
}

int main ()
{
    X* p;
    Y y;
    p = &y;
}

```

// Y 是 X 的一个子类

// p 是指向基类 X 对象的指针

// p 也可以指向 Y 的对象

因此，如果  $p$  的类型是  $X^*$ （“类型  $X$  的指针”），那么  $p$  也可以指向类型是  $X$  的子类的任何对象。然而，即使  $p$  正指向一个子类  $Y$  的一个实例，它的类型仍是  $X^*$ 。因此表达式，如  $p \rightarrow f()$  将调用在基类中定义的函数  $f()$ 。

回忆起  $p \rightarrow f()$  是  $( * p ) . f()$  的一个替代符号。它调用  $p$  指向的对象的成员函数  $f()$ 。但是如果  $p$  正指向  $p$  指的类的一个子类的一个对象  $y$  又会如何？如果子类  $Y$  有自己的  $f()$  重载模式呢？ $X::f()$  或  $Y::f()$  中会执行哪一个  $f()$ ？答案是  $p \rightarrow f()$  将执行  $X::f()$ ，因为  $p$  的类型是  $X^*$ 。与  $p$  此时正巧指向子类  $Y$  的一个实例这个事实并没有什么关系；正是  $p$  的静态定义的类型  $X^*$  决定了它的行为。

### 例 12.10 使用虚函数

这个演示程序声明  $p$  为一个指向基类  $X$  的对象的指针。首先它把  $p$  指向类  $X$  的一个实例  $x$ ；然后它把  $p$  指向派生类  $Y$  的一个实例  $y$ 。

```

class X
{
public:
    void f () { cout << "X:: f () executing \n"; }
};

class Y : public X
{
public:
    void f () { cout << "Y:: f () executing \n"; }
};

int main ()
{
    X x;
    Y y;
    X* p = &x;
    p -> f ();           // 调用 X:: f (), 因为 p 的类型为 X*
    p = &y;
    p -> f ();           // 调用 Y:: f (), 因为 p 的类型为 X*
}

X:: f ()  executing
Y:: f ()  executing

```

函数调用  $p \rightarrow f()$  进行了两次。两次都调用了相同模式的定义在基类  $X$  中的  $f()$ ，因为  $p$  声明为指向  $X$  对象的一个指针。让  $p$  指向  $y$  对第二次调用  $p \rightarrow f()$  并没有影响。

通过在它的声明中加入关键词 `virtual` 把  $X::f()$  变为一个虚函数：

```
class X
{
public:
    virtual void f() { cout << "X::f() executing\n"; }
};
```

其他的代码不变，结果现在变成了：

```
X::f() executing
Y::f() executing
```

此时第二次调用  $p \rightarrow$  调用了  $Y::f()$ ，而不是  $X::f()$ 。

这个例子展示了多态性：相同的调用  $p \rightarrow f()$  却调用了不同的函数。函数根据  $p$  指向的对象类而定，这叫做动态绑定，因为对要执行的实际代码的调用的组合（也就是绑定）被推迟到运行时间。指针的静态定义的类型决定了哪个成员函数被调用，这个规则由于把成员函数声明为 `virtual` 而被否决了。

下面是一个更加现实的例子。

### 例 12.11 通过虚函数实现多态性

下面是一个 `Person` 类、一个 `Student` 子类和 一个 `Professor` 子类：

```
class Person
{
public:
    Person(char * s) { name = new char[strlen(s) + 1]; strcpy(name, s); }
    void print() { cout << "My name is " << name << ".\n"; }
protected:
    char * name;
};

class Student : public Person
{
public:
    Student(char * s, float g) : Person(s), gpa(g) {}
    void print()
    { cout << "< My name is " << name << " and my G.P.A. is "
      << gpa << ".\n"; }
private:
    float gpa;
};

class Professor : public Person
{
public:
    Professor(char * s, int n) : Person(s), pubs(n) {}
    void print()
    { cout << "My name is " << name
      << " and I have " << pubs << " publications.\n"; }
private:
    int pubs;
};
```

```

    int pub's;
};

int main ()
{
    Person * p;
    Person x ("Bob");
    p = &x;
    p->print ();
    Student y ("Tom", 3.47);
    p = &y;
    p->print ();
    Professor z ("Ann", 7);
    p = &z;
    p->print ();
}

```

```

My name is Bob.
My name is Tom.
My name is Ann.

```

在基类中定义的 `print ()` 函数不是 `virtual` 型的。因此，调用 `p->print ()` 总是调用相同基类的函数 `Person::print ()`，因为 `p` 的类型是 `Person*`。在编译的时候，指针 `p` 静态地绑定于基类的函数。

现在把基类函数 `Person::print ()` 变为一个虚函数，运行相同的程序：

```

class Person
{
public:
    Person (char* s) { name = new char [strlen (s)+1]; strcpy (name, s); }
    virtual void print () { cout << "My name is " << name << ".\n"; }
protected:
    char* name;
};

```

```

My name is Bob.
My name is Tom and my G.P.A. is 3.47
My name is Ann and I have 7 publications.

```

此时指针 `p` 动态绑定于它指向的无论任何对象的函数 `print ()`。因此，第一次调用 `p->print ()` 调用了基类函数 `Person::print ()`，第二次调用了派生类函数 `Student::print ()`，第三次调用了派生类函数 `Professor::print ()`。称调用 `p->print ()` 是多态的，因为它的意义根据环境而变化。

一般来说，一个成员函数应该在如下情况时被声明为虚函数：当至少有几个它的子类需要定义这个函数它们自身的局部模式时。

## 12.8 虚拟析构函数

虚函数被具有相同符号的定义在子类中的函数重载。由于构造函数和析构函数的名字包含了它们的不同类的名字，看起来构造函数和析构函数是不能声明为 `virtual` 型的。对构造函数确实如此，然而对析构函数却有例外。

每个类都有唯一的析构函数，或在类定义中明确定义，或由编译器模糊定义。一个明确的析构函数可以定义为 virtual 型。下面的例子说明了定义在一个虚拟析构函数中的值。

### 例 12.12 内存泄露

本例的程序与例 12.6 类似。

```
class X
{
public:
    X () { p = new int [2]; cout << "X () . "; }
    ~X () { delete [] p; cout << "~X () . \n"; }
private:
    int * p;
};

class Y : public X
{
public:
    Y () { q = new int [1023]; cout << "Y (): Y:: q = " << q << ". "; }
    ~Y () { delete [] q; cout << "~Y () . "; }
private:
    int * q;
};

int main ()
{
    for (int i = 0; i < 8; i++)
    {
        X* r = new Y;
        delete r;
    }
}

X () . Y ():: Y:: q = 0x5821c. ~X () .
X () . Y ():: Y:: q = 0x5921c. ~X () .
X () . Y ():: Y:: q = 0x5a21c. ~X () .
X () . Y ():: Y:: q = 0x5b21c. ~X () .
X () . Y ():: Y:: q = 0x5c21c. ~X () .
X () . Y ():: Y:: q = 0x5d21c. ~X () .
X () . Y ():: Y:: q = 0x5e21c. ~X () .
X () . Y ():: Y:: q = 0x5f21c. ~X () .
```

For 循环的每一次重复都创建了一个动态的对象。如例 12.6 一样，构造函数按由上到下的顺序调用；首先是 X ()，然后是 Y ()，分配了 4100 字节的存储空间（每个整型用 4 个字节）。但是由于 r 被声明为一个指向 X 对象的指针，只有 X 的析构函数被调用，释放了仅仅 8 个字节。因此每次循环就损失了 4092 个字节。这个损失由指针 Y:: q 的实际值表明。

为了塞住这个泄露，把析构函数 ~X () 改为一个虚函数：

```
class X
{
public:
    X () { p = new int [2]; cout << "X () . "; }
    virtual ~X () { delete [] p; cout << "~X () . \n"; }
private:
    int * p;
};
```

```
};
X() . Y()::Y::q = 0x5a220. ~X() .
X() . Y()::Y::q = 0x5a220. ~X() .
X() . Y()::Y::q = 0x5a220. ~X() .
X() . Y()::Y::q = 0x5a220. ~X() .
X() . Y()::Y::q = 0x5a220. ~X() .
X() . Y()::Y::q = 0x5a220. ~X() .
X() . Y()::Y::q = 0x5a220. ~X() .
X() . Y()::Y::q = 0x5a220. ~X() .
```

由于基类的析构函数声明为 virtual 型，for 循环的每次重复调用了两个析构函数，因此，通过 new 操作符分配的内存重新存储起来。这允许指针 r 重用相同的内存。

这个例子解释了什么叫做“内存泄露”。在一个大规模的软件系统中，这可能导致灾难性的后果。此外，这个 bug 不易被找出来。规则是：当类层次结构中用到动态绑定时，把基类的析构函数声明为 virtual 型。

就像早期提到的那样，这些例子是设计来解释 C++ 的独特特征的，并不是为了标榜典型的编程例子的。

## 12.9 抽象基类

一个设计很好的面向对象程序应包括一个类的层次结构，它的内在关系可以用图 12.3 的树图描述：

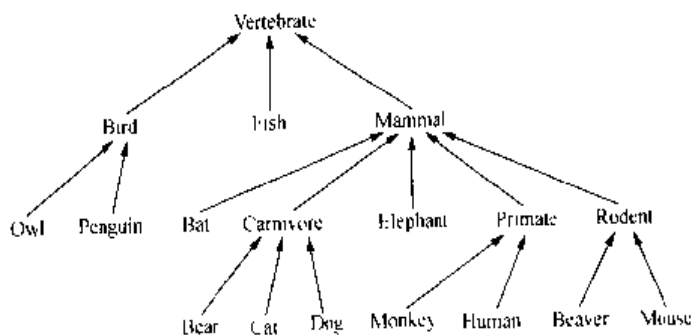


图 12.3 动物类的继承关系

这棵树的树叶上的类（例如 owl（猫头鹰），Fish（鱼类），Dog（狗））应包含专门的函数来实现它们特有类的行为（fish.swim（）<如鱼类游泳>，owl.fly<猫头鹰飞翔>，dog.dig<狗类挖掘>）。然而，这些函数中的一部分可能对一个类的所有子类都普遍适用（如 vertebrate.eat（）<脊椎动物吃>，mammal.suckle（）<哺乳动物吮吸>，primate.peel（）<灵长类动物的剥>）。这种函数应该在左类中声明为 virtual 型，然后为了特写的实现部分在它们的子类中重载。

如果一个虚函数肯定要在所有它的子类中重载，那么就没必要在它的所有基类中实现它。这可以通过把虚函数变为“pure”型来实现。一个纯虚函数就是在它的类中没有“实现”的虚函数。指定一个纯虚成员函数的语法是在初始化部分中插入“=0”来替代函数体，例如：

```
virtual int f () = 0;
```

在上面的脊椎动物类中，可能会确定 `eat ()` 函数在它的每个子类中重载，因此，在 `vertebrate` 脊椎动物基类中把它声明为一个纯虚函数：

```
class Vertebrate
{ public:
    virtual void eat () = 0;    //纯虚函数
};
class Fish : Public Vertebrate
{ public:
    void eat ();               //在其他地方的 Fish 类的特定实现
};
```

在一个类层次结构中的个体类，根据它们是否有任何纯虚成员函数把它们指定为或者“抽象的”或者“具体的”。一个抽象基类就是一个有一个或更多纯虚成员函数的类。一个具体的派生类就是一个没有任何纯虚成员函数的类。在下面的例子中，脊椎动物类是一个抽象基类，鱼类是一个具体的派生类。抽象基类不能实例化。

在一个类中一个纯虚函数的存在要求它的每个具体的派生类都要实现这个虚函数。在下面的例子中，如果方法 `vertebrate.eat ()`、`mammal.suckle ()` 和 `primate.peel ()` 是仅有的纯虚函数，那么抽象基类（“ABCs”）就是 `vertebrate`（脊椎动物），`mammal`（哺乳动物），`primate`（灵长类动物），其他的 15 个类就是具体的派生类（“CDCs”）。15 个具体派生类的每一个都会有它自己的 `eat ()` 函数的实现，哺乳动物类的 11 个具体派生类都会有它们自己的 `suckle ()` 函数的实现，灵长类动物的两个具体派生类都会有它们自己的 `peel ()` 函数实现。

抽象基类典型地在创建一个类的层次结构过程的第一阶段定义。它制订出框架，细节在抽象基类的子类中由框架产生。它的纯虚函数描述了在这个层次结构中的某种一致。

### 例 12.13 Media 类的层次结构

图 12.4 是类的一个层次结构，它描述了各种各样的媒体对象：

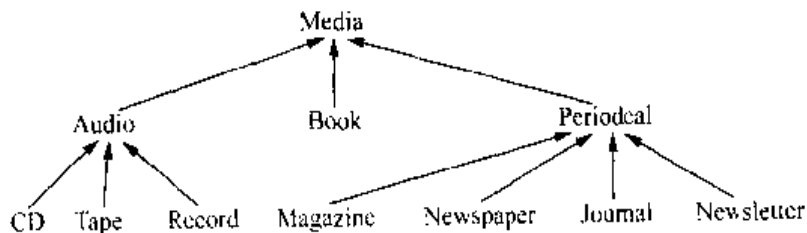


图 12.4 类 Media 的层次关系

主要的抽象基类是 `Media`（媒体）类：

```
class Media
{ public:
    virtual void print () = 0;
    virtual char* id () = 0;
protected;
```



```

        string title;
};

```

它有两个纯虚函数和一个数据成员。

下面是具体的 Book (书) 类:

```

class Book : Media
{
public:
    Book (string a = "", string t = "", string p = "", string i = "")
        : author (a), publisher (p), isbn (i) { title = t; }
    void print () { cout << title << " by " << author << endl; }
    char * id () { return isbn; }
private:
    string author, publisher, isbn;
};

```

它用自己的数据成员实现了两个虚函数。

下面是具体的 CD (光盘) 类:

```

class CD : Media
{
public:
    CD (string t = "", string c = "", string m = "", string n = "")
        : composer (c), make (m), number (n) { title = t; }
    void print () { cout << title << ", " << composer << endl; }
    char * id () { return make + " - " + number; }
private:
    string composer, make, number;
};

```

CD 类也是 Audio 类的一个具体派生类, Audio 类是另一个抽象基类。因此当 Audio 类定义时, 它的纯虚函数将会在 CD 类中实现。

下面是具体的 Magazine (杂志) 类:

```

class Magazine : Media
{
public:
    Magazine (string t = "", string i = "", int v = 0, int n = 0)
        : issn (i), volume (v), number (n) { title = t; }
    void print ()
    {
        cout << title << " Magazine, Vol. "
            << volume << ", No. " << number << endl;
    }
    char * id () { return issn; }
private:
    string issn, publisher;
    int volume, number;
};

```

Magazine 类也是 Periodical 类的一个具体派生类, Periodical 类是另一个抽象基类。因此当 Periodical 类定义时, 它的纯虚函数将会在 Magazine 中实现。

下面是上面定义的四类的一个测试驱动程序:

```

int main ()

```

```

Book book ("Bjarne Stroustrup", "The C++ Programming Language",
    "Addison-Wesley", "0-201-53992-6");
Magazine magazine ("TIME", "0040-781X", 145, 23);
CD cd ("BACH CANTATAS", "Johann Sebastian Bach",
    "ARCHIV", "D120541");
book.print();
cout << "\tid: " << book.id() << endl;
magazine.print();
cout << "\tid: " << magazine.id() << endl;
cd.print();
cout << "\tid: " << cd.id() << endl;

```

```

The C++ Programming Language by Bjarne Stroustrup
id: 0-201-53992-6
TIME Magazine, Vol. 145, No.23
id: 0040-781X
BACH CANTATAS, Johann Sebastian Bach
Id: ARCHIV D120541

```

注意到所有对函数 `print()` 和 `id()` 的调用都和它们类的实现独立。因此，这些函数的实现可以改变而不用对程序作任何修改。例如，可以把 `Book.print()` 函数改为：

```

void print()
{ cout << title << " by " << author
    << ".\nPublished by " << publisher << ".\n";
}

```

得到结果是：

```

The C++ Programming Language by Bjarne Stroustrup
Published by Addison-Wesley.

```

而程序没任何变化。

## 12.10 面向对象程序设计

“面向对象程序设计”指的是派生类和虚函数的使用。面向对象程序设计的详细内容不属于本书的范围内容。可以查看附录 H 中的列出的书目 [Bergin]、[Perry] 和 [Wang]。

假设有三部电视机，每部都配备有它自己的视频盒式录像机 (VCR)。像大多数的视频盒式录像机一样，有它自己的特征和令人迷惑的操作手册。这三个视频盒式录像机各不相同，需要不同的复杂操作来使用它们。然后有一天，出现了一种简单的可以操作所有类型视频盒式录像机的遥控器。例如，它有一个简单的“RECORD”按钮，它能使任何视频盒式录像机把当前的电视节目录到当前的磁带上。这个神奇的设备描述了面向对象程序设计 (OOP) 的精华：通过一个简单的接口对不同的实现进行概念上的简化。在这个例子中，接口就是遥控器，实现就是遥控器和单个视频盒式录像机的 (隐蔽) 操作，VCR 执行所要求的功能 (“RECORD”，“STOP”，“PLAY”等等)。接口就是下面的抽象基类：

```
class VCR
```

```
    public:  
        virtual void on () = 0 ;  
        virtual void off () = 0 ;  
        virtual void record () = 0 ;  
        virtual void stop () = 0 ;  
        virtual void play () = 0 ;  
};
```

实现是下面的具体的派生类:

```
class Panasonic : public VCR {  
public:  
    void on () ;  
    void off () ;  
    void record () ;  
    void stop () ;  
    void play () ;  
};  
  
class Sony : public VCR {  
public:  
    void on () ;  
    void off () ;  
    void record () ;  
    void stop () ;  
    void play () ;  
};  
  
class Mitsubishi : public VCR {  
public:  
    void on () ;  
    void off () ;  
    void record () ;  
    void stop () ;  
    void play () ;  
};
```

面向对象系统的一个很重要的优势就是可扩展性。这是指系统很容易被扩展。在下面的例子中, 如果视频盒式录像机 (VCR) 能和将来可能增加的新的视频盒式录像机工作一致的话, 那么这个 VCR 控制器是“可扩展的”。当 VCR 系列被扩展了以后, 增加了一个东芝的或用一个 RCA 替换了索尼的, 控制器不应该改变。

在面向对象的程序设计中, 设想有两种截然不同的系统角度: 消费者的角度 (也就是顾客或用户) 是要做什么, 生产厂商 (也就是服务者或实现者) 的角度是如何做。消费者只看到抽象基类, 而生产厂商看到了具体的派生类。和生产厂商相对的是, 顾客的行为通常叫做操作, 而生产厂商的行为通常叫做方法。在 C++ 中, 动作就是纯虚函数, 方法就是在具体派生中对它们的实现。在这个上下文中, 抽象基类 (从用户的角度) 叫做系统接口, 具体的派生类 (从实现者的角度) 叫做实现。如表 12.1 所示。

表 12.1 面向对象程序设计中的两个方面

| 系统接口    | 系统实现     |
|---------|----------|
| (用户的角度) | (实现者的角度) |
| 告诉做了什么  | 告诉如何做    |
| 抽象基类    | 具体派生类    |
| 操作      | 方法       |
| 纯虚函数    | 函数       |

当用到指向对象的指针时,如例 12.13,两分法最有效。然后可以利用动态绑定使系统接口与系统实现更独立。可扩展性由于只有新增加的方法才能被编译这个事实而更容易了。

## 复 习 题

- 12.1 组合和继承的区别是什么?
- 12.2 `protected` 和 `private` 的区别是什么?
- 12.3 在一个继承的层次结构中,默认的构造函数和析构函数是如何区分的?
- 12.4 什么是虚成员函数?
- 12.5 什么是纯虚成员函数?
- 12.6 什么是内存泄露?
- 12.7 虚拟析构函数如何塞住内存泄露?
- 12.8 什么是抽象基类?
- 12.9 什么是具体的派生类?
- 12.10 静态绑定和动态绑定的区别是什么?
- 12.11 什么是多态性?
- 12.12 多态性如何提高可扩展性?
- 12.13 下面的定义有什么错误?

```

class X
{
protected:
    int a;
};

class Y : public X
{
public:
    void set (X x, int c) { x.a = c; }
};

```

## 习 题

- 12.1 为玩牌者实现一个 `Card` 类, 一个组合的 `Hand` 类和一个组合的 `Deck` 类。
- 12.2 实现图 12.5 所示的类层次结构。

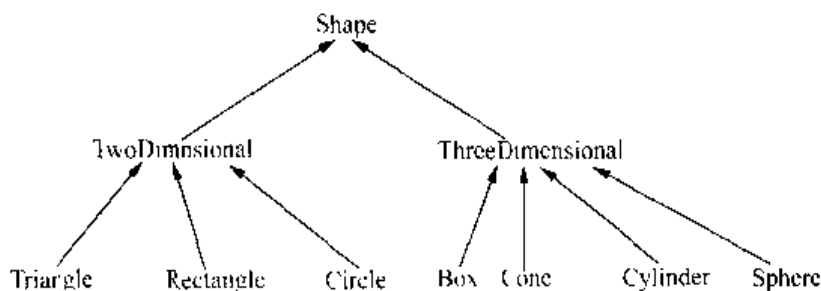


图 12.5 类的层次关系

12.3 定义并测试一个 Name 类，它的对象如图 12.6 所示。然后改变 Person 类，使得 name 的类型是 Name 而不是 string。

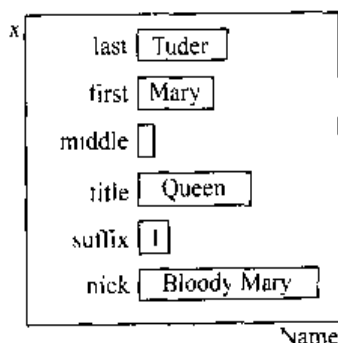


图 12.6 Name 类的对象

## 复习题答案

- 12.1 类的组合指的是用一个类去声明另一个类的成员。继承是指从基类派生出一个子类。
- 12.2 一个 private 型成员在它的类定义外部的任何地方都不能被访问。一个 protected 型成员在它的类定义外部的任何地方都不能被访问，例外的是在派生类的定义中它是可访问的。
- 12.3 在一个类的继承层次结构中，每个默认的构造函数在执行之前调用它的父默认构造函数，每个默认的析构函数在它执行之后调用它的父默认析构函数。结果是所有的父默认构造函数按由上到下的顺序执行，所有的父默认析构函数按由下到上的顺序执行。
- 12.4 虚成员函数就是一个在子类中可以重载的成员函数。
- 12.5 纯虚成员函数就是不能直接调用的虚成员函数，只有它在派生类中的重载函数可以被调用。纯虚函数通过在声明中的初始化的最后的“=0”来标识。
- 12.6 内存泄露就是在程序中由于调用错误的析构函数而引起的对内存的失败访问。见例 12.12。
- 12.7 通过声明一个基类的析构函数为 virtual 型的，就可以避免例 12.12 中所示的内存泄露，因为在析构函数被调用后，它指定的子类的析构函数也会被调用。

- 12.8 抽象基类就是一个至少包括一个纯虚函数的基类。抽象基类不能实例化。
- 12.9 一个具体的派生类就是一个能被实例化的抽象基类的子类。也就是说，是一个不包括纯虚函数的子类。
- 12.10 静态绑定指的是在编译时把一个成员函数调用与函数本身连接起来，与之相反，动态绑定是直到运行时连接起来。C++ 中通过使用虚函数并把指针传给对象才使动态的绑定变为可能。
- 12.11 多态性指的是运行时的绑定，它是在有虚函数的类中用到指向对象的指针时发生的。表达式  $p \rightarrow f()$  将调用定义在  $p$  指向的对象中的函数  $f()$ 。然而，这个对象可以属于子类序列中的任一个，可以在运行时做出了类的选择。如果基类函数是虚的，那么在运行时做出调用哪个  $f()$  的选择（绑定）。因此，表达式  $p \rightarrow f()$  可以采取“多种形式”。
- 12.12 多态性允许在一个类的层次结构中加入新的子类和方法，而不必更改已应用这个层次结构接口的应用程序，这样就提高了可扩展性。
- 12.13 `protected` 型数据成员  $a$  只有当它是当前对象的成员时（也就是只有它是  $this \rightarrow a$  时）才能访问。Y 的任何对象  $x$  都不能访问  $x.a$ 。

## 习题答案

### 12.1 首先实现一个 Card 类：

```
enum Rank { TWO=2, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN,
            JACK, QUEEN, KING, ACE };
enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES };

class Card
{
    friend class Hand;
    friend class Deck;
    friend ostream& operator<< (ostream&, const Card&);
public:
    char rank () { return rank_; }
    char suit () { return suit_; }
private:
    Card () {}
    Card (Rank rank, Suit suit) : rank_ (rank), suit_ (suit) {}
    Card (const Card& c) : rank_ (c.rank_), suit_ (c.suit_) {}
    ~Card () {}
    Rank rank_;
    Suit suit_;
};
```

这个类用枚举类型来描述一副纸牌的 13 个等级和 4 组可能的纸牌。如果希望实现 `Hand` 和 `Deck` 类，则在这里把它们声明为 `Card` 类的友元类。这将使它们能够访问 `Card` 类的私有成员。

注意，所有的三个构造函数和析构函数都声明为私有型的。这将避免除了它的两个友元外，任何 Card 的创建和销毁。

下面是 Cards 的重载插入运算符 << 的实现：

```
ostream& operator << (ostream& ostr, const Card& card)
{
    switch (card.rank_)
    {
        case TWO      : ostr << "two of ";      break;
        case THREE    : ostr << "three of ";     break;
        case FOUR     : ostr << "four of ";      break;
        case FIVE     : ostr << "five of ";      break;
        case SIX      : ostr << "six of ";       break;
        case SEVEN    : ostr << "seven of ";     break;
        case EIGHT    : ostr << "eight of ";     break;
        case NINE     : ostr << "nine of ";      break;
        case TEN      : ostr << "ten of ";       break;
        case JACK     : ostr << "jack of ";      break;
        case QUEEN    : ostr << "queen of ";     break;
        case KING     : ostr << "king of ";      break;
        case ACE      : ostr << "ace of ";       break;
    }

    switch (card.suit_)
    {
        case CLUBS    : ostr << "clubs";        break;
        case DIAMONDS : ostr << "diamonds";     break;
        case HEARTS   : ostr << "hearts";       break;
        case SPADES   : ostr << "spades";       break;
    }

    return ostr;
}
```

下面是 Hand 类的实现：

```
#include "Card.h"
class Hand
{
    friend class Deck;
public:
    Hand (unsigned n=5) : size (n) { cards = new Card [n]; }
    ~Hand () { delete [] cards; }
    void display ();
    int isPair ();
    int isTwoPair ();
    int isThreeOfKind ();
    int isStraight ();
    int isFlush ();
    int isFullHouse ();
    int isFourOfKind ();
    int isStraightFlush ();
private:
    unsigned size;
    Card * cards;
    void sort ();
};
```

它用一个数组存储手中的纸牌。Sort () 函数是一个 Deck 类在处理完 hand 后调用的

私有应用程序。它可以用任何像冒泡排序法这样简单的排序算法实现。Display () 函数也是简单易懂，它使用了在 Card 类重载的插入运算符 <<。

八个布尔型函数不是那么简单易懂，它们标识了特殊的纸牌 hands。下面是 isThreeOfKind () 函数的实现：

```
int Hand::isThreeOfKind ()
{ if (cards [0].rank_ == cards [1].rank_
    && cards [1].rank_ == cards [2].rank_
    && cards [2].rank_ != cards [3].rank_
    && cards [3].rank_ != cards [4].rank_) return 1;
  if (cards [0].rank_ != cards [1].rank_
    && cards [1].rank_ == cards [2].rank_
    && cards [2].rank_ == cards [3].rank_
    && cards [3].rank_ != cards [4].rank_) return 1;
  if (cards [0].rank_ != cards [1].rank_
    && cards [1].rank_ != cards [2].rank_
    && cards [2].rank_ == cards [3].rank_
    && cards [3].rank_ == cards [4].rank_) return 1;
  return 0;
}
```

由于 hand 是由 rank\_ 排序的，因此有三张同样大小牌与两张不同大小牌的形式只能是：x x x y z, x y y y z 或 x y z z z。如果这三种形式中任何一种被发现，函数返回 1，否则返回 0。

is Pair () 函数，is Two Pair () 函数，is Full House () 函数及 is Four Of Kind () 函数相似。

is Straight () 函数、is Flush () 函数及 is Straight Flush () 函数也很巧妙。这里是 is Flush () 函数：

```
int Hand::isFlush ()
{ for (int i = 1; i < size; i++)
    if (cards [i].suit_ != cards [0].suit_) return 0;
  return 1;
}
```

这里将每一个 suit\_ 与它下一个的 suit\_ 进行比较直到第 5 张 (card [1] 到 card [4])，如果这些比较有任何一个不一样，我们就知道 hand 不是同花并返回 0。如果循环正常结束，则 4 对都符合，于是返回 1。

这里是 Deck 类：

```
#include "Random.h"
#include "Hand.h"
class Deck
{ public:
    Deck ();
    void shuffle ();
    void deal (Hand&, unsigned = 5);
private:
    unsigned top;
```



```

    Card cards [52];
    Random random;
};

```

它在洗牌中使用了 `Random` 类。注意到这个 `random` 对象声明为私有型的，是因为它只有另一个成员函数使用。

```

void Deck:: deal (Hand& hand, unsigned size)
{ for (int i = 0; i < size; i++)
    hand.cards [i] = cards [top++];
  hand.sort ();
}

```

`top` 成员总是定位一副牌的顶，也就是下一张将要发的牌。因此 `deal ()` 函数复制 `deck` 的最上面 5 张牌到手中的 `cards` 数组。接着它将 `hand` 排序。

`Deck` 的构造函数初始化一副牌中的 52 张牌，采用顺序：梅花 2，梅花 3…黑桃 A。

```

Deck:: Deck ()
{ for (int i = 0; i < 52; i++)
    { cards [i].rank_ = Rank (i%13);
      cards [i].suit_ = Suit (i%4);
    }
  top = 0;
}

```

所以如果手上的牌没先洗牌就发了，第一个手上将是 2 到 6 的梅花同花顺。

最后，这里是 `shuffle ()` 函数：

```

void Deck:: shuffle ()
{ for (int i = 0; i < 52; i++) //做 52 个随机交换
    { int j = random.integer (0, 51);
      Card c = cards [i];
      cards [i] = cards [j];
      cards [j] = c;
    }
  top = 0;
}

```

它将 52 张牌中的每一张与从另一张随机选出的牌相交换。

## 12.2 下面是抽象基类：

```

const double PI = 3.14159265358979;
class Shape
{ public:
    virtual void print () = 0;
    virtual float area () = 0;
};
class TwoDimensional : public Shape
{ public:
    virtual float perimeter () = 0;
};
class ThreeDimensional : public Shape
{ public:
    virtual float volume () = 0;
};

```

```
};
```

注意, `print()` 函数和 `area()` 函数的原型在这个类层次结构中对所有的类都相同, 因此它们的接口(纯虚函数)放在了 `Shape` 基类中。但只有二维的 `shape`(形状)才有周长, 只有三维的 `shape`(形状)才有容积, 因此它们的接口放在了相应的第二层抽象基类中。

下面是七个具体派生类中的两个:

```
class Circle : public TwoDimensional
{
public:
    Circle(float r) : radius(r) {}
    void print() { cout << "Shape is a circle. \n"; }
    float perimeter() { return 2 * PI * radius; }
    float area() { return PI * radius * radius; }
private:
    float radius;
};

class Cone : public ThreeDimensional
{
public:
    Cone(float r, float h) : radius(r), height(h) {}
    void print();
    float area();
    float volume() { return PI * radius * radius * height/3; }
private:
    float radius, height;
};

void Cone::print()
{ cout << "Cone: radius = " << radius << ", height = "
  << height << endl;
}

float Cone::area()
{ float s = sqrt(radius*radius + height*height);
  return PI * radius * (radius + s);
}
;
```

其他五个具体派生相似。

### 12.3 下面是 `Name` 类的接口:

```
class Name
{
    friend ostream& operator<< (ostream&, const Name&);
    friend istream& operator>> (istream&, Name&);
public:
    Name(char*, char*, char*, char*, char*, char*);
    string last() { return last_; }
    string first() { return first_; }
    string middle() { return middle_; }
    string title() { return title_; }
    string suffix() { return suffix_; }
    string nick() { return nick_; }
    void last(string s) { last_ = s; }
    void first(string s) { first_ = s; }
    void middle(string s) { middle_ = s; }
```

```

    void title (string s) { title_ = s; }
    void suffix (string s) { suffix_ = s; }
    void nick (string s) { nick_ = s; }
    void dump ();
private:
    string last_ , first_ , middle_ , title_ , suffix_ , nick_;
};

```

下面是 Name 类的一个实现:

```

Name::Name (char * last="", char * first="", char * middle="",
            char * title="", char * suffix="", char * nick="")
: last_ (last), first_ (first), middle_ (middle), title_ (title),
  suffix_ (suffix), nick_ (nick) {}

void Name::dump ()
{ cout << "\t\t Last Name: " << last_ << endl;
  cout << "\t\t First Name: " << first_ << endl;
  cout << "\t\t Middle Names: " << middle_ << endl;
  cout << "\t\t Title: " << title_ << endl;
  cout << "\t\t Suffix: " << suffix_ << endl;
  cout << "\t\t Nick name: " << nick_ << endl;
}

ostream& operator<< (ostream& out, const Name& x)
{ if (x.title_ != "") out << x.title_ << " ";
  out << x.first_ << " ";
  if (x.middle_ != "") out << x.middle_ << " ";
  out << x.last_ ;
  if (x.suffix_ != "") out << " " << x.suffix_ ;
  if (x.nick_ != "") out << ", \" " << x.nick_ << "\" ";
  return out;
}

istream& operator>> (istream& in, Name& x)
{ char buffer [80];
  in.getline (buffer, 80, '|');
  x.last_ = buffer;
  in.getline (buffer, 80, '|');
  x.first_ = buffer;
  in.getline (buffer, 80, '|');
  x.middle_ = buffer;
  in.getline (buffer, 80, '|');
  x.title_ = buffer;
  in.getline (buffer, 80, '|');
  x.suffix_ = buffer;
  in.getline (buffer, 80, '|');
  x.nick_ = buffer;
  return in;
}

```

最后, 这里是改进的 Person 类:

```

#include "date.h"
#include "Name.h"
class Person
{ public:

```

```

    Person (char * r: "", int s = 0, char * nat = "U.S.A.")
        : name (r), sex (s), nationality (:nat) {}
    void setDOB (int m, int d, int y) { dob.setDate (m, d, y); }
    void setDOD (int m, int d, int y) { dod.setDate (m, d, y); }
    void printName () { cout << name; }
    void printNationality () { cout << nationality; }
    void printDOB () { cout << dob; }
    void printDOD () { cout << dod; }
protected:
    Name name;
    Date dob, dod;           // 出生日期, 死亡日期
    int sex;                 // 0 = 女, 1 = 男
    string nationality;
};

```

下面是 Name 类的一个测试驱动程序:

```

#include <iostream.h>
#include "Name.h"

int main ()
{
    Name x ("Bach", "Johann", "Sebastian");
    cout << x << endl;
    x.dump ();
    x.last ("Clinton");
    x.first ("William");
    x.middle ("Jefferson");
    x.title ("President");
    x.nick ("Bill");
    cout << x << endl;
    x.dump ();
    cin >> x;
    cout << x << endl;
    cout << "x.last    = [" << x.last () << "]\n";
    cout << "x.first    = [" << x.first () << "]\n";
    cout << "x.middle   = [" << x.middle () << "]\n";
    cout << "x.title    = [" << x.title () << "]\n";
    cout << "x.suffix   = [" << x.suffix () << "]\n";
    cout << "x.nick    = [" << x.nick () << "]\n";
}

```

```

Johann Sebastian Bach
    Last Name: Bach
    First Name: Johann
    Middle Names: Sebastian
    Title:
    Suffix:
    Nickname:
President William Jefferson Clinton, "Bill"
    Last Name: Clinton
    First Name: William
    Middle Names: Jefferson
    Title: President
    Suffix:
    Nickname: Bill

```

```
Tudor|Mary||Queen|I|Bloody Mary
Queen Mary Tudor I, "Bloody Mary"
x.last   = [Tudor]
x.first  = [Mary]
x.middle = []
x.title  = [Queen]
x.suffix = [I]
x.nick   = [Bloody Mary]
```

## 第 13 章 模板与迭代符

### 13.1 引言

模板是一个产生具体代码的抽象处方。模板可用来产生函数和类。编译器用模板为各种各样的函数或类产生代码，就像用一个甜饼切割机把各种各样的生面团生成甜饼一样。用模板产生的实际的函数或类叫做那个模板的实例。

相同的模板可用来产生许多不同的实例。这可通过模板参数的方式来实现，它对模板的作用就如同普通参数对普通函数的作用一样。但普通参数是对象的替代者，而模板参数则是类型和类的替代者。

C++ 提供实例化模板，这种便利是它的主要特征之一，也是区别它与大部分编程语言的特征之一。作为一种自动生成代码的机制，为了提高编程效率，允许对它进行大量的改进。

### 13.2 函数模板

在很多排序算法中，需要交换一对元素。这种简单的任务通常由一个独立的函数来完成。例如，下面的函数交换整数：

```
void swap (int& m, int& n)
{   int temp = m;
    m = n;
    n = temp;
}
```

然而如果要对字符串对象排序，那么将需要一个不同的函数：

```
void swap (string& s1, string& s2)
{   string temp = s1;
    s1 = s2;
    s2 = temp;
}
```

这两个函数做了同一件事情。它们惟一的区别是所交换对象的类型。可以通过用一个函数模板替代这两个函数来避免这种冗余：

#### 例 13.1 swap 函数模板

```
template < class T >
```

```
void swap (T& x, T& y)
{   T temp = x;
    x = y;
    y = temp;
}
```

符号 T 叫做类型参数。它只是一个替代者，当函数被调用时，它被一个实际的类型或类替换。

一个函数模板可以如同一个普通的函数那样声明，除非下面这个声明优先于它：

```
template <class T>
```

类型参数 T 可以用来在函数定义内部替代普通的类型。这里单词 class 的意思是“任何类型”。更为一般地是，一个模板可以有几个类型参数，如下面的声明：

```
template <class T, class U, class V>
```

函数模板的调用方式和普通的函数一样：

```
int m = 22, n = 66;
swap (m, n);
string s1 = "John Adams", s2 = "James Madison";
swap (s1, s2);
Rational x (22/7), y (-3);
swap (x, y);
```

对每次调用，编译器产生完整的函数，用参数属于的类型或类替代类型参数。因此，如上所示的那样调用 swap (m, n) 生成了整型 swap 函数，调用 swap (s1, s2) 对字符串对象生成 swap 函数。

函数模板是对函数重载的一种直接的概括。可以写出 swap 函数的几种重载的版本，每一种都是可能需要的。简单的 swap 函数模板可以完成同样的任务。但它有两方面的改进。它只需被写一次来覆盖用它可能替代的不同类型，并不必提前决定将用它替代的类型；任何类型或类都能用来替代类型参数 T。函数模板在结构相似的函数群体中共享源代码。

### 例 13.2 冒泡排序法模板

下面是一个冒泡排序函数（如例 6.13）和一个任何基类向量的 print 函数。

```
template <class T>
void sort (T* v, int n)
{   for (int i = 1; i < n; i++)
        for (int j = 0; j < n - i; j++)
            if (v[j] > v[j+1]) swap (v[j], v[j+1]);
}

template <class T>
void print (T* v, int n)
{   for (int i = 0; i < n; i++)
        cout << " " << v[i];
    cout << endl;
```

```

;

int main ()
{ short a [9] = {55, 33, 88, 11, 44, 99, 77, 22, 66};
  print (a, 9);
  sort (a, 9);
  print (a, 9);
  string s [7] = {"Tom", "Hal", "Dan", "Bob", "Sue", "Ann", "Gus"};
  print (s, 7);
  sort (s, 7);
  print (s, 7);
}

```

这里的 `sort ()` 和 `print ()` 函数都是模板函数。在第一次调用时类型参数 `T` 被类型 `short` 替代，第二次调用类型参数 `T` 被类 `string` 替代。

一个函数模板的作用就像一个提纲。编译器使用模板产生所需要的每个函数的模式。在前面的例子中，编译器产生 `sort ()` 函数的两种版本和 `print ()` 函数的两种版本，每个函数中的一个为类型 `short` 的，一个是为类型 `string` 的。这种个体版本叫做函数模板的实例，产生它们的过程叫做实例化模板。一个模板实例的函数也叫做模板函数。使用模板就是一种自动代码生成的方式，它能使程序设计人员把更多的工作留给编译器。

### 13.3 类模板

除了产生的是类而不是函数外，一个类模板的作用和一个函数模板一样。一般的语法是：

```
template< class T, ... > class X { ... };
```

就像函数模板一样，一个类模板也可以有几个模板参数。此外，它们中的一部分可以是普通的非类型参数：

```
template< class T, int n, class U > class X { ... };
```

当然，由于模板是在编译时实例化的，传递给非类型参数的值必须是常量：

```

template< class T, int n>
class X { };

int main ()
{ X< float, 22 > x1;          //ok
  const int n = 44;
  X< char, n > x2;            //ok
  int m = 66;
  X< short, m > x3;           //错误：m 必须是常量
}

```

类模板有时也叫做参数化的类型。

一个类模板的成员函数本身也是函数模板，具有和它们的类相同的头文件。例如，定义



在类模板中的函数 f ()

```
template< class T>
class X
{ T square (T t) { return t * t; }
};
```

的作用和下面的模板函数的作用一样:

```
template< class T>
T square (T t) { return t * t; }
```

它由编译器实例化, 用传递给它的类型替换模板参数 T。因此, 声明

```
X< short> x;
```

生成了类和对象。

```
class X_short
{ short square (short t) { return t * t; }
};
X_short x;
```

除非是编译器可能为这个类用了某个名字而不是 X\_short。

### 例 13.3 一个 Stack 类模板

栈是一种简单的数据结构, 它模拟一种普通的相同类型对象的栈 (例如碟子组成的栈), 限制条件是对象只能在栈顶进行插入和删除。换句话说, 栈是一种只能在一端访问的线性数据结构。一个栈类通过隐蔽数据结构的实现部分来抽象这种定义, 它只能以公有函数的方式访问, 这个函数模拟了上面所描述的限制操作。

下面是生成 Stack 类的一个类模板:

```
template < class T>
class Stack
{ public:
    Stack (int s = 100) : size (s), top (-1) { data = new T [size]; }
    ~Stack () { delete [] data; }
    void push (const T& x) { data [++top] = x; }
    T pop () { return data [top--]; }
    int isempty () const { return top == -1; }
    int isfull () const { return top == size - 1; }
private:
    int size;
    int top;
    T* data;
};
```

定义使用了一个数组 data 来实现一个栈。构造函数初始化了这个数组的大小, 把类型为 T 的许多元素分配给了这个数组, 并初始化它的 top 指针为 -1。top 的值总是比栈中的元素个数少, 除非栈是空的, 这时 top 是栈中的头元素在这个数组中的下标。push () 函数把一个对象插入栈中, pop () 从栈中把一个对象删除掉。当它的 top 值为 -1 时, 一个栈为

isEmpty(), 当它的 top 指针的值为 size-1 时, 栈是 isFull()。

下面是一个测试栈模板的程序:

```
int main ()
{
    Stack<int> intStack1 (5);
    Stack<int> intStack2 (10);
    Stack<char> charStack (8);
    intStack1.push (77);
    charStack.push ('A');
    intStack2.push (22);
    charStack.push ('E');
    charStack.push ('K');
    intStack2.push (44);
    cout << intStack2.pop () << endl;
    cout << intStack2.pop () << endl;
    if (intStack2.isEmpty ()) cout << "intStack2 is empty. \n";
}
```

44

22

intStack2 is empty.

模板有一个参数 T, 它用来指定存储在栈中的对象的类型。第一行声明了 intStack1 为一个能容纳 5 个整数的栈。与此类似, intStack2 是一个能容纳 10 个整数的栈, charStack 是一个能容纳 8 个字符的栈。

把对象压入栈中和弹出栈中后, 最后一行调用了 intStack2 的 isEmpty() 函数。此时, 这两个栈类和三个栈对象如图 13.1 所示:

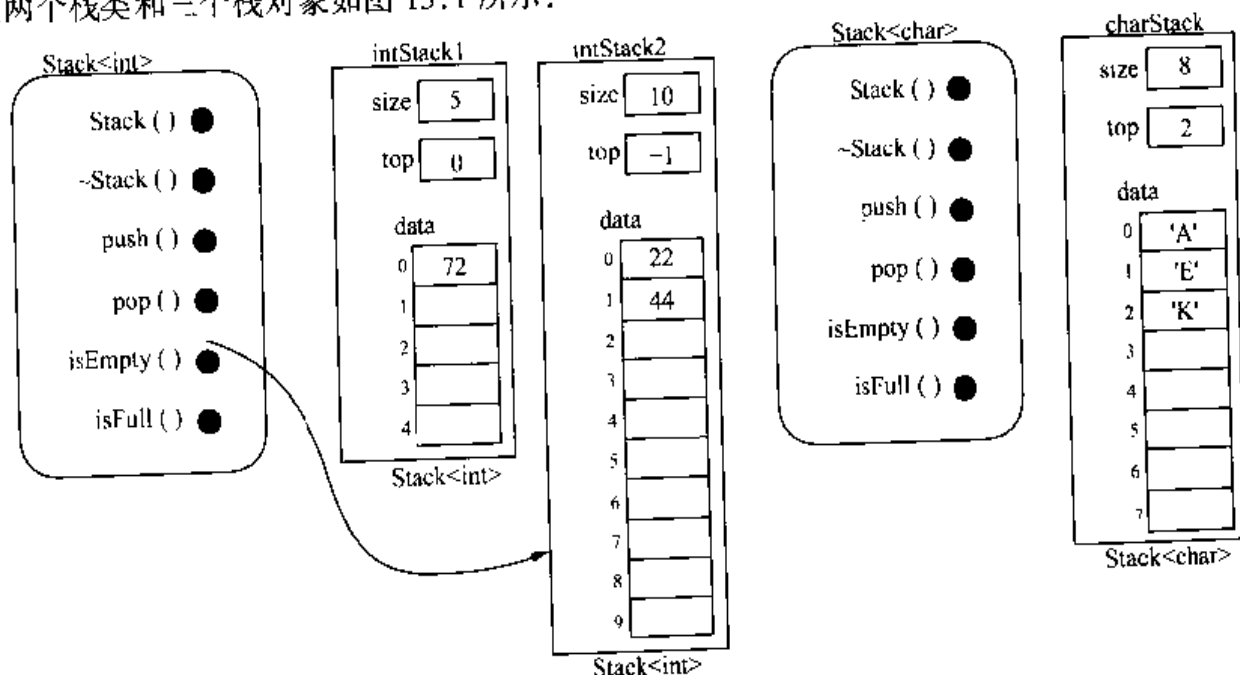


图 13.1 模板 Stack 生成的类

调用 instack2.isEmpty() 返回值 1 (也就是“true”), 因为此时 intStack2.top 的值为 -1。注意有两个栈类模板的实例: Stack<int> 和 Stack<char>。它们是截然不同的类, 每

个都由编译器生成。每个类有它自己的六个不同的成员函数。例如，这两个函数 `Stack<int>::pop()` 和 `Stack<char>::pop` 是不同的：一个返回一个整数，一个返回一个字符。

## 13.4 容器类

一个容器类就是一个简单的可以容纳其他对象的对象。普通的数组和栈是容器类。一个容器类就是一个其实例是容器的类。例 13.3 中的类 `Stack<int>` 和 `Stack<char>` 是容器类。类模板是一种很自然的生成容器类的机制，因为被容纳的对象类型可以用一个模板参数来指定。

如果一个容器的所有对象的类型都相同，它就叫做同类容器；否则它就叫做异类容器。栈，数组等等都是同类容器。

一个向量就是一个相同类型对象的下标序列。这个单词是从数学上借来的，它原指一个三维的点  $x = (x_1, x_2, x_3)$ 。当然，那只是一个有三个实数的排列。除了在 C++ 中这些值必须从 0 开始外，组合中的下标和数组中的下标值一样，由于下标不能在源代码中书写，因此，用符号括弧 `[]` 代替。`x[0]` 表示 `x1`，`x[1]` 表示 `x2`，`x[2]` 表示 `x3`。

### 例 13.4 一个向量类模板

```
template< class T>
class Vector
{ public:
    Vector (unsigned n=8) : size (n), _data (new T [size]) {}
    Vector (const Vector< T> &v) : size (v.size), _data (new T [size])
        { copy (v); }
    ~Vector () { delete [] _data; }
    Vector< T> & operator = (const Vector< T> &);
    T& operator [] (unsigned i) const { return _data [i]; }
    unsigned size () { return size; }
protected:
    T* _data;
    unsigned size;
    void copy (const Vector< T> &);
};

template< class T>
Vector< T> & Vector< T>::operator = (const Vector< T> &v)
{ size = v.size;
  _data = new T [size];
  copy (v);
  return *this;
}

template< class T>
```

```

void Vector<T>::copy (const Vector<T> &v)
{ unsigned min_size = (size < v.size ? size : v.size);
  for (int i=0; i<min_size; i++)
    data[i] = v.data[i];
}

```

注意相同的模板指示符 `template<class T>` 必须在一个成员函数的每个实现前，它也优先于类的声明：`template<class T>`。

这个模板将允许下面的代码：

```

Vector<short> v;
v[5] = 127;
Vector<short> w = v, x(3);
cout << w.size();

```

这里的 `v` 和 `w` 都是有 8 个 `short` 型元素的向量对象，`x` 是有图 13.2 所示的 3 个 `short` 型元素的向量对象。这个类和它的三个对象可以用图 13.2 的图形形象化表示。它显示的是成员函数 `w.size()` 执行时的情形。类 `Vector<short>` 已从模板中实例化，三个对象 `v`、`w` 和 `x` 已被这个类实例化。注意 `copy()` 函数是一个保护型的公用程序函数，因此它不能被任何类的实例调用。

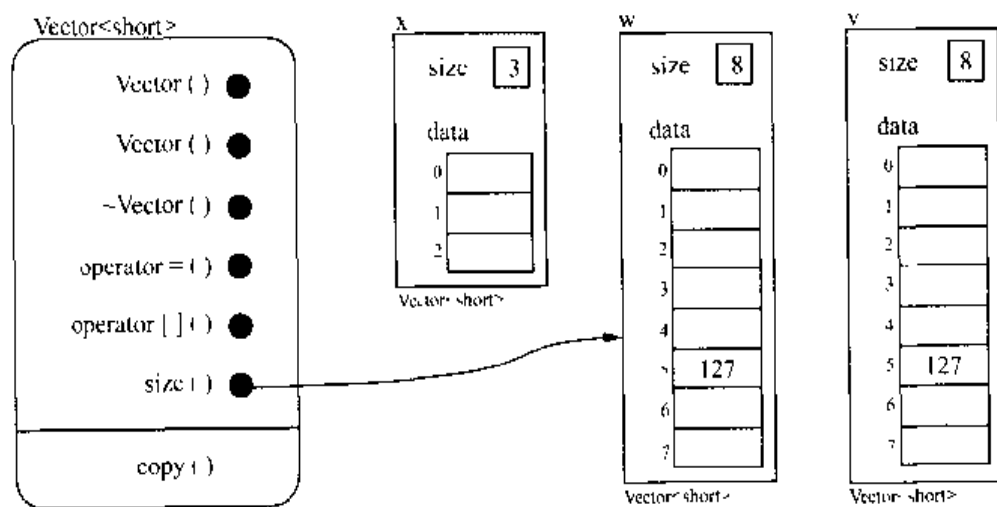


图 13.2 Vector 模板生成的类

注意表达式 `v[5]` 用在了赋值语句的左边，尽管这个表达式是一个函数调用。这是可能的，因为下标运算符返回一个 `Vector<T>` 的引用，使它成了一个左值。

类模板也叫做参数化的类型，因为它们就像参数可以传递给的类型。例如，上面的对象 `b` 的类型是 `Vector<double>`，因此元素类型 `double` 就像模板 `Vector<T>` 的一个参数。

## 13.5 子类模板

类模板的继承和普通的类继承的功能一样。为了阐明这种技巧，将定义一个在例 13.4

中定义的 Vector 类模板的子类模板。

### 例 13.5 一个向量的子类模板

像例 13.4 中模板实现的那样，向量类的一个问题是它要求以 0 开始的下标，也就是说，所有的下标都必须以 0 开始。这是 C++ 自身的要求。一些其他的编程语言允许数组下标从 1 或者任何整数开始，通过声明一个子类模板可以把这个有用的特征加入到 Vector 模板中：

```
template < class T>
class Array : public Vector< T>
{ public:
    Array (int i, int j) : i0 (i), Vector< T> (j - i + 1) ;
    Array (const Array< T> & v) : i0 (v.i0), Vector< T> (v) ;
    T& operator [] (int i) const { return Vector< T> :: operator [] (i - i0); }
    int firstSubscript () const { return i0; }
    int lastSubscript () const { return i0 + _size - 1; }
protected:
    int i0;
};
```

这个 Array 类模板继承了 Vector 类模板的所有功能并且允许下标从任何整数开始。列出的第一个成员函数允许用户在对象声明时指定下标的第一个和最后一个值。第二个函数是这个子类的复制构造函数，第三个函数是重载的下标运算符。最后两个函数只是返回下标范围的第一个和最后一个值，如图 13.3 所示。

注意这两个 Array 构造函数如何调用相应的 Vector 构造函数，Array 下标运算符如何调用 Vector 下标运算符。

下面是一个测试驱动程序和一个例子的运行：

```
#include <iostream>
#include "Array.h"

int main ()
{ Array< float> x (1, 3);
  x [1] = 3.14159;
  x [2] = 0.08516;
  x [3] = 5041.92;
  cout << "x.size () = " << x.size () << endl;
  cout << "x.firstSubscript () = " << x.firstSubscript () << endl;
  cout << "x.lastSubscript () = " << x.lastSubscript () << endl;
  for (int i = 1; i <= 3; i++)
    cout << "x [" << i << "] = " << x [i] << endl;
}
```

```
x.size () = 3
x.firstSubscript () = 1
x.lastSubscript () = 3
x [1] = 3.14159
x [2] = 0.08516
x [3] = 5041.92
```

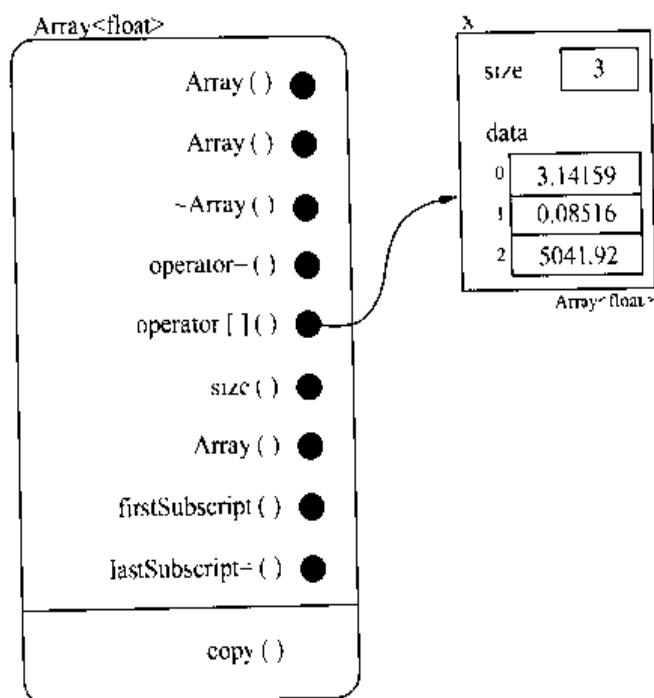


图 13.3 Array 模板产生的类

## 13.6 把模板类传到模板参数

前面已经见过把一个类传递给一个模板参数的例子：

```
Stack<Rational> s;           // Rational 对象的一个栈
Vector<string> a;           // string 对象的一个向量
```

由于模板类和普通类相似，我们也可以把它们传递给模板参数：

```
Stack<Vector<int>> s;        // Vector 对象的一个栈
Array<Stack<Rational>> a;   // Stack 对象的一个数组
```

### 例 13.6 一个 Matrix 类模板

一个 Matrix 本质上是一个二维的向量。例如，一个  $2 \times 3$  矩阵就是一个 2 行 3 列的表：

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

可以把它认为是一个有 2 个元素的向量，每个元素是一个有 3 个元素的向量：

$$\left[ [a \ b \ c] [d \ e \ f] \right]$$

这一观点的好处是它允许重用 Vector 类模板来定义一个新的 Matrix 类模板。

为了有利于内存的动态分配，把矩阵定义为一个指向向量的指针的向量：

```
Vector<Vector<T>*>
```

把一个类模板指针传递给了由外部的尖括号标定的模板参数。这实际上意味着当 `Matrix` 类模板被实例化时，结果类的实例将包括指向向量的指针的向量。

```
template<class T>
class Matrix
{ public:
    Matrix (unsigned r=1, unsigned c=1) : row (r)
    { for (int i = 0; i<r; i++) row [i] = new Vector<T> (c); }
    ~Matrix () { for (int i=0; i < row.size (); i++) delete row [i]; }
    Vector<T> & operator [] (unsigned i) const { return * row [i]; }
    unsigned rows () { return row.size (); }
    unsigned columns () { return row [0] -> size (); }
protected:
    Vector< Vector< T > * > row;
};
```

这里仅有的数据成员是 `row`——一个指向向量的指针的向量。作为一个向量，`row` 能和下标运算符一起使用，`row [i]` 返回一个表示矩阵第 *i* 行的向量的指针。

默认的构造函数给每个 `row [i]` 赋予一个新的包含 *c* 个类型为 *T* 的元素的向量。析构函数不得不单独地删除这些向量中的每一个。`row ()` 和 `columns ()` 函数返回矩阵中的行和列的数目。行的数目是 `Vector< Vector< T > * >` 对象 `row` 的成员函数 `size ()` 返回的值。列的数目是 `Vector< Vector< T > * >` 对象 `* row [0]` 的成员函数 `size ()` 返回的值，它可以被 `(* row [0]).size ()` 或者被 `row [0] -> size ()` 引用。

下面是一个测试驱动程序和一个例子的运行：

```
int main ()
{ Matrix<float> a (2, 3);
  a [0] [0] = 0.0; a [0] [1] = 0.1; a [0] [2] = 0.2;
  a [1] [0] = 1.0; a [1] [1] = 1.1; a [1] [2] = 1.2;
  cout << "The matrix a has " << a.rows () << " rows and "
        << a.columns () << " columns. \n";
  for (int i=0; i<2; i++)
    { for (int j=0; j<3; j++)
        cout << a [i] [j] << " ";
      cout << endl;
    }
}
```

```
The matrix a has 2 rows and 3 columns:
0 0.1 0.2
1 1.1 1.2
```

矩阵 *a* 可以形象化表示如图 13.4 所示：

这个图显示了其中一个下标访问调用 `a [1] [2]` 过程时的情形。

注意，实际的成员值 0.2, 1.1, 等等。存储在两个单独的 `Vector<float>` 对象中。`Matrix<float>` 对象 *m* 只包含这些对象的指针。

注意，`Matrix` 类模板对 `Vector` 类模板使用了组合，而 `Array` 类模板对 `Vector` 类模板使用了继承。

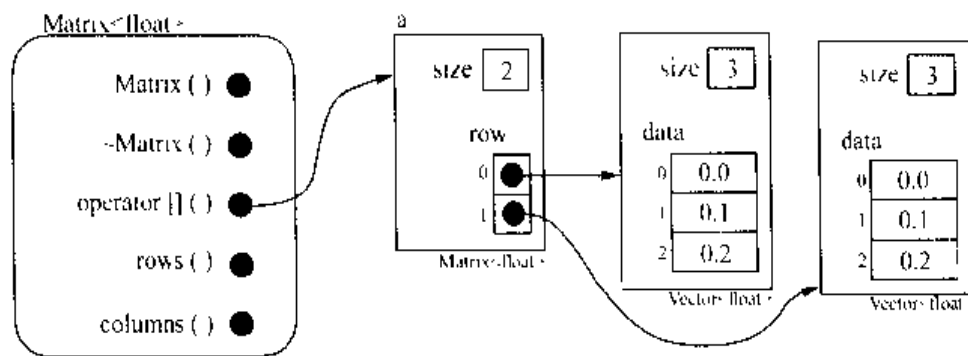


图 13.4 Matrix 模板产生的类

## 13.7 链表的一个类模板

在例 10.13 中曾引入了链表。为了动态存储的便利，这种数据结构提供了向量的一个替代。也就是说，与向量不同，链表能根据正被存储的数据成员的数目来动态地扩大和缩小。链表中的未使用的元素没有废弃的空间。

### 例 13.7 一个 list 类模板

一个链表由一个结点的链接序列组成。每个结点包含一个数据成员和一个到下个结点的链接。因此可从由定义一个 ListNode 类模板开始：

```
template < class T>
class ListNode
{
    friend class List< T>;
public:
    ListNode (T& t, ListNode< T> * p) : data (t), next (p) {}
protected:
    T data;           // data 字段
    ListNode * next;  // 指向链表中的下个结点
};
```

构造函数创建了一个新结点，把 T 的值 t 赋给了 data 字段，把指针 p 赋给 next 字段，如图 13.5 所示：

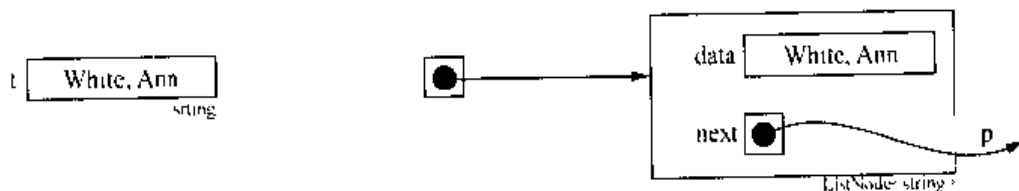


图 13.5 ListNode 模板产生的类

如果 T 是一个类（而不是一个普通的类型），它的构造函数将被 data 的声明调用。

注意，这里的类 List< T> 被声明为 ListNode 类的一个友元。这将允许 List 类的成员函



数访问 Node 类的保护型成员，对于要编译的语句，一些编译器要求下面的前向参考优先于 ListNode 模板定义：

```
template< class T>
class List;
```

这只是简单地告诉编译器，标识符 List 将在后面作为一个类模板定义。

下面是 List 类模板接口，它紧随 ListNode 模板定义：

```
template< class T>
class List
{ public:
    List () : first (0) {} ,
    ~List ();
    void insert (T t);           // 把 t 插入到链表的头部
    int remove (T& t);          // 删除链表中的第一个元素 t
    bool isEmpty () { return (first == 0); }
    void print ();
protected:
    ListNode<T> * first;
    ListNode<T> * newNode (T& t, ListNode<T> * p)
        { ListNode<T> * q = new ListNode<T> (t, p); return q; }
};
```

一个 List 对象只包含一个叫做 first 的指针。它指向一个 ListNode 对象。默认的构造函数把指针初始化为 NULL。在元素已被插入到链表中后，first 指针将指向链表的第一个元素，如图 13.6 所示。

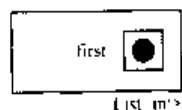


图 13.6 List 模板产生的类

newNode 函数调用 new 运算符通过 ListNode () 构造函数的方式获得一个新的 ListNode 对象。新的结点将在它的 data 字段中包含 T 的值 t，在它的 next 字段中包含指针 p。newNode 函数被声明为保护型的，因为它是一个只被其他成员函数使用的公用程序函数。

List 的析构函数负责删除链表中的所有元素。

```
template< class T>
List<T>::~~List ()
{ ListNode<T> * temp;
  for (ListNode<T> * p = first; p; )           // 遍历链表
  { temp = p;
    p = p->next;
    delete temp;
  }
}
```

这要在一个遍历链表的循环中完成。每个结点通过对指向结点的指针调用 delete 运算符删除。

insert () 函数创建了一个新结点，它包含 T 的值 t，然后，把这个新结点插入到链表的开始：

```
template< class T>
```

```
void List<T>::insert (T t)
{ ListNode<T> * p = newNode (t, first);
  first = p;
}
```

由于新结点将成为链表的第一个结点，它的 next 指针应该指向目前的链表中第一个结点。把 first 指针传递给 NewNode 构造函数就做到了这一点。然后，first 指针被重新设置为指向新的结点。

remove () 函数从链表中去掉第一个元素，通过参数 t 中的调用返回它的 data 值。根据操作是否成功，函数的返回值将是 1 或 0：

```
template< class T>
int List<T>::remove (T& t)
{ if (isEmpty ()) return 0;    // 标记没有删除的标号
  t = first->data;             // 调用返回的 data 值
  ListNode<T> * p = first;
  first = first->next;         // 推进 first 指针来删除结点
  delete p;
  return 1;                   // 标记成功删除的标号
}
```

print () 函数只是遍历链表，打印每个结点的 data 值：

```
template< class T>
void List<T>::print ()
{ for (ListNode<T> * p = first; p; p = p->next)
  cout << p->data << " -> ";
  cout << "\n";
}
```

下面是一个测试驱动程序和一个例子的运行：

```
#include <iostream.h>
#include "List.h"

int main ()
{ List<string> friends;
  friends.insert ("Bowen, Van");
  friends.insert ("Dixon, Tom");
  friends.insert ("Mason, Joe");
  friends.insert ("White, Ann");
  friends.print ();
  string name;
  friends.remove (name);
  cout << "Removed: " << name << endl;
  friends.print ();
}
```

```
White, Ann -> Mason, Joe -> Dixon, Tom -> Bowen, Van -> *
Removed: White Ann
Mason, Joe -> Dixon, Tom -> Bowen, Van -> *
```

注意，由于每个元素在链表的开始插入，在结束时它们与插入时顺序相反。



- 改变存储在当前位置的数据值;
- 决定是否在迭代符的当前位置实际上有一个元素;
- 在容器中推进到下一个位置。

由于这五个操作应由每个迭代符实现, 用这些函数声明一个抽象基类是有意义的。实际上就需要一个抽象基类模板, 因为容器类将是模板的实例:

```
template< class T>
class Iterator
{ public:
    virtual void reset () = 0;           //初始化迭代符
    virtual T operator () () = 0;        //读当前值
    virtual void operator = (T t) = 0;   //写当前值
    virtual int operator! () = 0;         //决定是否元素存在
    virtual int operator ++ () = 0;       //向下一个元素推进
};
```

记起每个纯虚函数原型是由关键字“virtual”开始, 由代码“( ) = 0”结束。圆括号是必须的, 因为它是一个函数, initializer “= 0”使它成为一个纯虚函数。也要记起一个抽象基类是一个包含至少一个纯虚函数的任何类 (参见 12.9 节)。

现在可以使用这个抽象基类模板来为各种各样的容器类派生迭代符模板。

例 13.7 中的 List 类模板有一个明显的缺点: 它只允许在链表的前面进行插入和删除。如下面的例子所示, 一个链表迭代符将解决这个问题。

### 例 13.8 List 类模板的一个 Iterator 类模板

```
#include "List.h"
#include "Iterator.h"

template< class T>
class ListIter : public Iterator< T>
{ public:
    ListIter (List< T> &l) : list (l) { reset (); }
    virtual void reset () { previous = NULL; current = list.first; }
    virtual T operator () () { return current -> data; }
    virtual void operator = (T t) { current -> data = t; }
    virtual int operator! ();           // 决定是否当前存在迭代符
    virtual int operator ++ ();          // 向下一个元素推进
    void insert (T t);                  // 在当前的元素后插入 t
    void preInsert (T t);                // 在当前的元素前插入 t
    void remove ();                     // 去掉当前的元素
protected:
    ListNode< T> * current;              // 指向当前的结点
    ListNode< T> * previous;             // 指向前面的结点
    List< T> & list;                     // 这是正被遍历的链表
};
```

除了一个构造函数和五个基本操作外, 还加入了三个其他的将会使链表更有用的函数。允许在链表中的任何地方插入和删除元素。

`operator!()` 函数有两个目的。首先, 如果必要的话, 它重新设置 `current` 指针, 然后它报告返回这个指针是否为空。第一个目的是, 在调用 `remove()` 函数后的“清洗”, `remove()` 函数删除 `current` 指向的结点。

```
template< class T>
int ListIter< T>:: operator! ()
{
    if (current == NULL)          // 重新设置 current 指针
        if (previous == NULL) current = list.first;
        else current = previous->next;
    return (current != NULL); // 如果 current 存在, 返回 TRUE
}
```

如果 `current` 和 `previous` 指针都是空的, 那么链表是空的, 或者它只有一个元素。因此设置 `current` 等于表的 `first` 指针将或者使 `current` 为空, 或者让它指向链表的第一个元素。如果 `current` 是空的, 但是 `previous` 正指向一个结点, 那么简单地重新设置 `current` 指向紧随那个结点的元素。最后, 函数根据 `current` 是否为空返回 0 或者 1。这允许函数以下面的形式被调用:

```
if (! it) ...
```

此处的 `it` 是一个迭代符。表达式 `!it` 读作“一个当前元素存在”, 因为如果 `current` 是非空, 函数将返回 1 (也就是“true”)。在调用一个要使用这个指针的插入或删除函数前, 我们用这个函数来检查 `current` 指针的状态。

`operator++()` 通过在推进迭代符的 `previous` 指针前把链表中迭代符的 `current` 指针推进到下一个元素来“增加”迭代符。如果它发现 `current` 指针为空, 它就用 `operator!()` 函数执行相同的重新设置程序来处理这个操作:

```
template< class T>
int ListIter< T>:: operator++ ()
{
    if (current == NULL)          // 重新设置 current 指针
        if (previous == NULL) current = list.first;
        else current = previous->next;
    else
    {
        previous = current; // 推进 current 指针
        current = current->next;
    }
    return (current != NULL);      //如果 current 指针存在, 返回 TRUE
}
```

这个运算符允许对链表的简单遍历:

```
for (it.reset(); !it; ++ it) ...
```

就像一个普通的 `for` 循环遍历一个数组一样, 它重新设置迭代符来定位链表中的第一个元素。然后在访问那个元素后, 增加迭代符来推进和访问下一个元素。只要 `!it` 返回“true”循环就继续, 这意味着仍有一个元素要访问。

`insert()` 函数为 `t` 创建一个新的结点, 然后, 立即把那个结点插入到 `current` 结点后:

```

template< class T>
void ListIter<T>::insert (T t)
{
    ListNode<T> * p = list.newNode (t, 0);
    if (list.isEmpty ()) list.first = p;
    else
    {
        p->next = current->next;
        current->next = p;
    }
}

```

insert 操作可以形象化如图 13.8 所示。

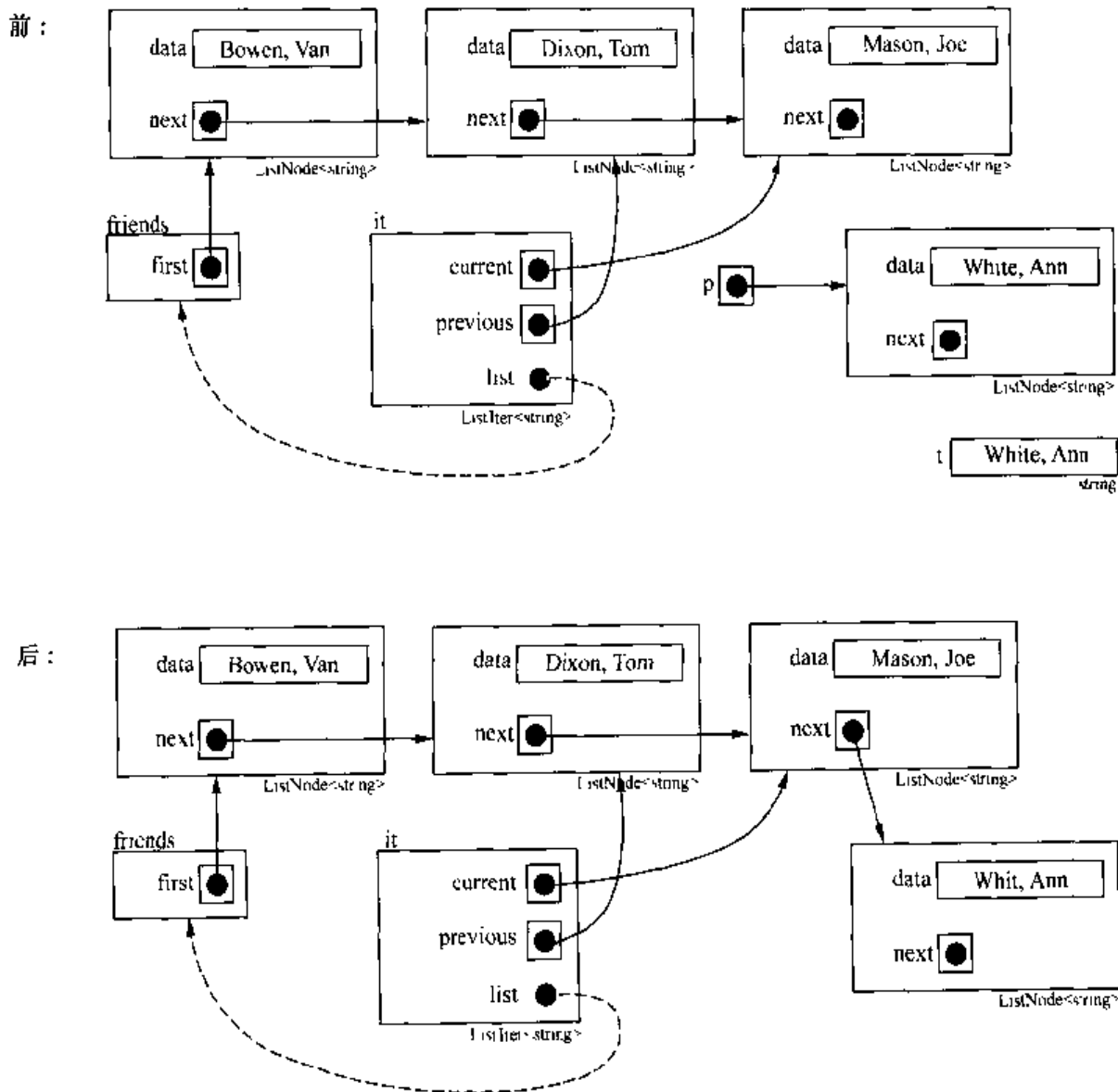


图 13.8 insert 操作示例

注意，操作使 current 指针和 previous 指针不变。

除了把新结点插入到 current 结点前外，preInsert () 函数与 insert () 函数相似：

```

template< class T>
void ListIter<T>::preInsert (T t)

```

```

}
ListNode<T> * p = list.newNode (t, current);
if (current == list.first) list.first = previous = p;
else previous -> next = p;
}

```

preInsert 操作可以形象化如图 13.9 所示。

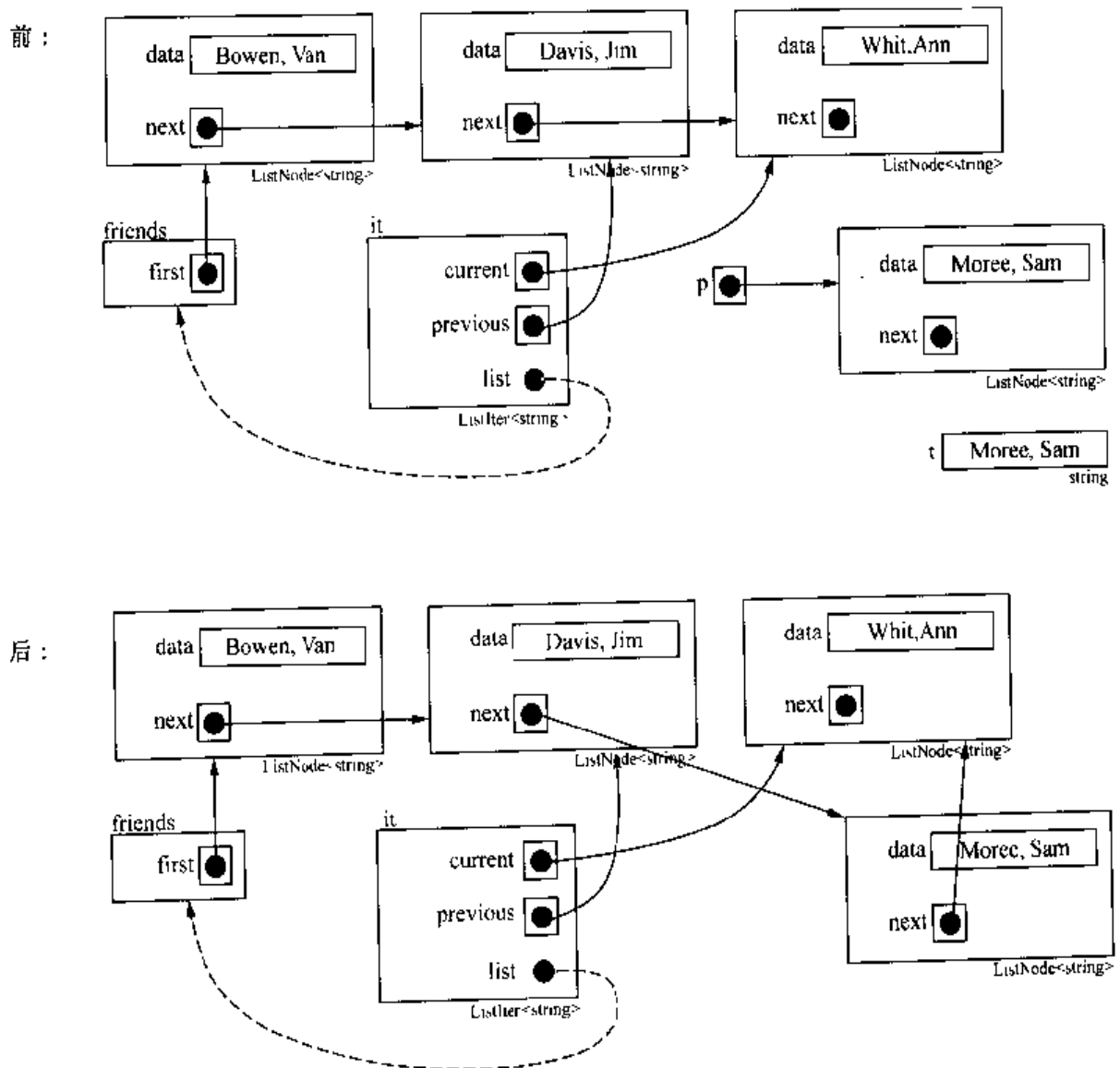


图 13.9 preInsert 操作示例

注意，就像 insert 一样，这个操作也使 current 指针和 previous 指针不变。

remove () 函数删除 current 结点：

```

template< class T>
void ListIter<T>::remove ()
{ if (current == list.first) list.first = current -> next;
  else previous -> next = current -> next;
}

```

```

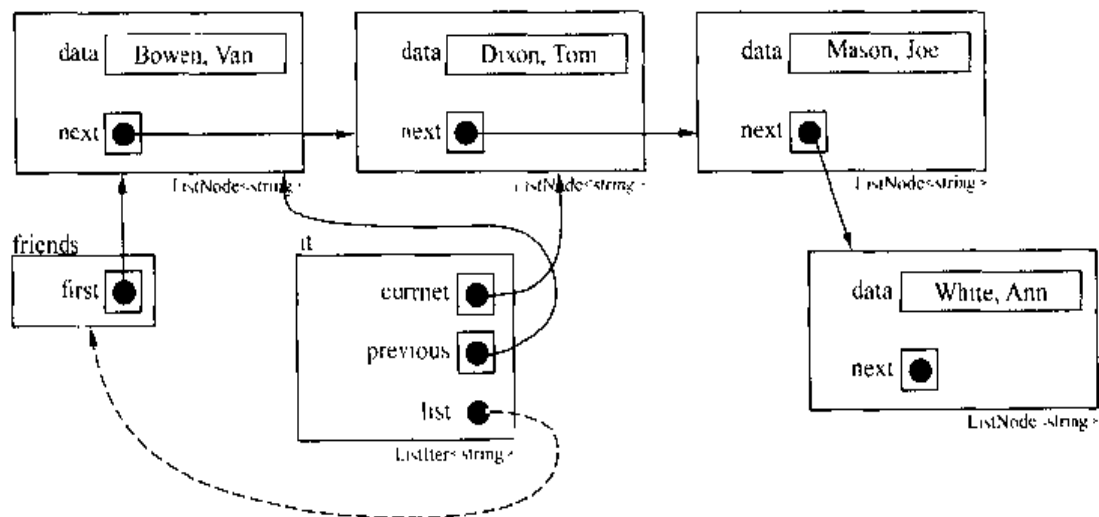
delete current;
current = 0;
}

```

它使 previous 指针不变, current 指针为空。

remove 操作可以形象化如图 13.10 所示。

前：



后：

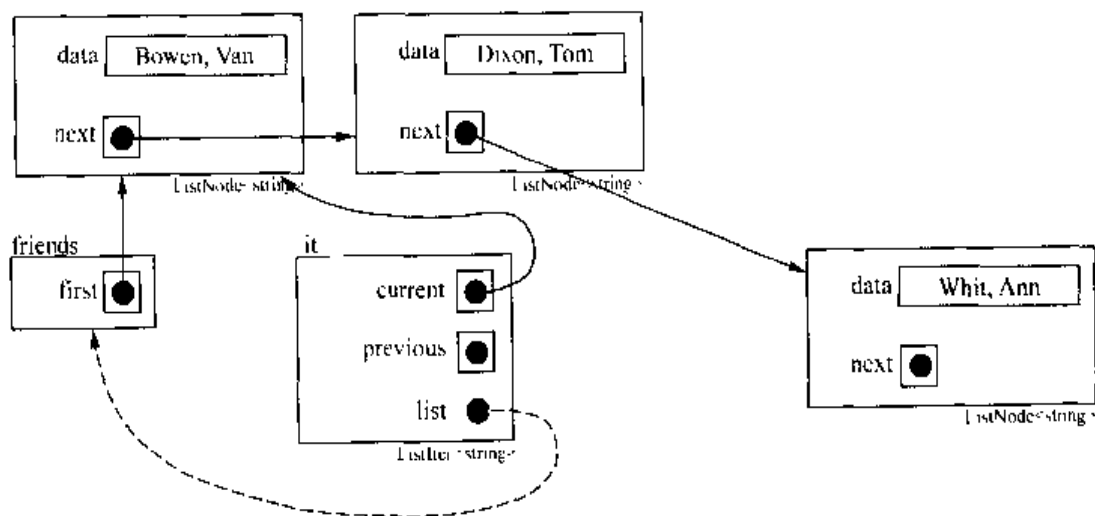


图 13.10 remove 操作示例

下面是链表迭代符的一个测试驱动程序：

```

#include "ListIter.h"
int main ()
{ List< string> friends;
  ListIter< string> it (friends);
  it.insert ("Bowen, Van");
  ++ it;                //设置 current 指向第一个结点
}

```



```

it.insert ("Dixon, Tom");
++ it;           //设置 current 指向第二个结点
it.insert ("Mason, Joe");
++ it;           //设置 current 指向第二个结点
it.insert ("White, Ann");
++ it;           //设置 current 指向第四个结点
friends.print ();
it.reset ();      //设置 current 指向第一个结点
++ it;           //设置 current 指向第二个结点
it = "Davis, Jim"; //用新名字替换
++ it;           //设置 current 指向第三个结点
it.remove ();     //删除第三个结点
friends.print ();
if (! it) it.preInsert ("Morse, Sam");
friends.print ();
for (it.reset (); ! it; ++ it) //遍历全部链表
    it = "[" + it () + "]";
friends.print ();

```

```

Bowen, Van -> Dixon, Tom -> Mason, Joe -> White, Ann -> *
Bowen, Van -> Davis, Jim -> White, Ann -> *
Bowen, Van -> Davis, Jim -> Morse, Sam -> White, Ann -> *
[Bowen, Van] -> [Davis, Jim] -> [Morse, Sam] -> [White, Ann] -> *

```

for 循环通过为每个字符串前加一个左括弧和后加一个右括弧来改变链表中的每个 data 值。注意，赋值语句 `it = "[" + it () + "]"` 调用了 `ListIter<string>` 类的 `operator ()()` 函数和 `operator =()` 函数，还有定义在 `string` 类中的构造函数 `string (const char *)` 和 `operator +()` 函数。

为了让 `ListIter` 对象访问需要用来完成它们的工作的 `List` 对象的保护型成员，需要声明 `ListIter` 类作为 `List` 类的一个友元：

```

template< class T>
class List
{
    friend class ListIter< T>;
public:
    //其他成员
protected:
    ListNode< T> * first;
    //其他成员
};

```

`List` 迭代符也需要访问 `ListNode` 对象的保护型成员：

```

Template< class T>
Class ListNode
{
    friend class List< T>;
    friend class ListIter< T>;
public:
    ListNode (T& t, ListNode< T> * p) : data (t), next (p) {}
protected:
    T data; //data 字段

```

```

    ListNode* next;           //指向链表中的下个结点
};

```

一个循环就像一个窗口，允许一次访问容器中的一个元素。循环有时也叫做光标，因为它们定位在整个结构中的一个元素，就像计算机屏幕上的光标定位一个字符位置一样。

一个结构可以有超过一个的迭代符。例如，人们可以如下在一个链表中声明三个迭代符：

```

List< float > list;
ListIter< float > it1 (list), it2 (list), it3 (list);
it1.insert (11.01);
++ it1;
it1.insert (22.02);
++ it1;
it1.insert (33.03);
for (it2.reset (); ! it2; ++ it2)
it2 = 10 * it2;           //用 10 去乘每个存储的数字
it3 = it1;                //在第一个元素中用 330.3 替代 110.1

```

迭代符是相互独立的。当 it2 遍历链表时，it1 仍固定在第三个元素上。

## 复 习 题

- 13.1 一个函数模板和一个模板函数的区别是什么？
- 13.2 一个类模板和一个模板类的区别是什么？
- 13.3 使用一个链表替代一个向量的好处和缺点是什么？
- 13.4 一个迭代符和一个数组下标如何相像？

## 习 题

- 13.1 写出并测试一个程序，它初始化一个函数模板，这个函数模板返回两个值中最小值的。
- 13.2 写出并测试一个程序，它初始化一个函数模板，这个函数模板对一个排序的对象数组实现折半查找。
- 13.3 实现并测试一个产生 Queue 类的模板。一个 queue 就像一个栈，除了插入是在线性结构的末尾进行和删除在其他的末尾进行外。它模仿一个普通的等待队列。
- 13.4 更改 Vector 类模板，使得已知的向量能改变它们的大小。
- 13.5 为 Vector 类模板增加一个构造函数，它把一个普通的数组取代为向量。
- 13.6 从 Vector< T> 类模板派生一个 Array< T, E> 类模板，此处的第二个模板参数 E 有一个枚举类型用来作为数组的下标。

## 复习题答案

- 13.1 一个函数模板就是一个模板，它用来产生函数。一个模板函数就是一个函数，它由一

个模板产生。例如，例 13.1 中的 `swap (T&, T&)` 函数是一个函数模板，但调用 `swap (m, n)` 产生了实际的模板函数，它被这次调用调用。

- 13.2 一个类模板就是一个模板，它用来产生类。一个模板类就是一个类，它由一个模板产生。例如，13.3 中的 `Stack` 就是一个类模板，但声明中用到的类型 `Stack < int >` 实际上是一个模板类。
- 13.3 向量有通过下标运算符的方式直接访问（也叫做“随机访问”）它们的元素的优势。因此，如果元素按顺序保存，可以使用折半查找算法很快找到它们。链表的优势是动态的，因此它们从未使用比当前所需要更多的空间，并且它们不受预先决定的大小的限制（除了计算机的内存大小外）。因此向量有时间优势，链表有空间优势。
- 13.4 迭代符和数组下标的作用都像一个数据结构的定位器。下面的代码显示了它们的作用是相同的：

```
float a [100];           // 一个大小为 100 的浮点型数组
int i = 0;               // 数组的下标
a [i] = 3.14159;
for (i = 0; i < 100; i++) cout << a [i];
List<float> list;        // 一个浮点型链表
ListIter<float> it (list); // 链表的一个迭代符
it = 3.14159;
for (it.reset(); ! it; ++it) cout << it ();
```

## 习 题 答 案

- 13.1 一个最小化函数应该比较两个相同类型的对象，并返回值较小的那个元素。类型应该是模板参数 `T`：

```
template<class T>
T min (T x, T y)
{ return ( x < y ? x : y);
}
```

这个实现使用了条件表达式运算符：`( x < y ? x : y)`。如果 `x` 小于 `y`，表达式的值为 `x`；否则它的值为 `y`。

下面是测试驱动程序和一个简单的运行：

```
# include "Ratio.h"
int main ()
{ cout << "min (22, 44) = " << min (22, 44) << endl;
  cout << "min (66.66, 33.33) = " << min (66.66, 33.33) << endl;
  Ratio x (22, 7), y (314, 100);
  cout << "min (x, y) = " << min (x, y) << endl;
```

```
min (22, 44)    = 22
min (66.66, 33.33) = 33.33
min (x, y)     = 314/100
```

- 13.2 一个查找函数应该传递给数组 `a`、要找到对象 `key`、数组的定义查找范围下标的范围。如果找到了对象，它在数组中的下标应该返回；否则，函数应该返回 `-1` 来标志对象没有找到：

```
template <class T>
int search (T a [], T key, int first, int last)
{ while (first <= last)
    { int mid = (first + last) / 2;
      if (key < a [mid]) last = mid - 1;
      else if (key > a [mid]) first = mid + 1;
      else return mid;
    }
  return -1; //未找到
}
```

在 `while` 循环的内部，从 `a [first]` 到 `a [last]` 的子列被 `mid` 折半分开。如果 `key < a [mid]`，那么，在数组的第二部分不能找到 `key`，因此 `last` 被重新设置为 `mid - 1` 来把查找的范围减少到第一部分。否则，如果 `key > a [mid]`，那么 `key` 不可能在数组的第一部分，因此 `first` 被重新设置为 `mid + 1` 来把查找的范围减少到第二部分。如果两种条件都是错误的，那么 `key == a [mid]`，就可以返回了。

下面是测试驱动程序和一个简单的运行：

```
template <class T> int search (T [], T, int, int)

string names []
= {"Adams", "Black", "Cohen", "Davis", "Evans", "Frost",
   "Greer", "Healy", "Irwin", "Jones", "Kelly", "Lewis"};

int main ()
{ string name;
  while (cin >> name)
  { int location = search (names, name, 0, 9);
    if (location == -1) cout << name << " is not in the list.\n";
    else cout << name << " is in position " << location << endl;
  }
}
```

```
Green
Green is in position 6
Black
Black is in position 1
White
White is not in list.
Adams
Adams is in position 0
Jones
Jones is in position 9
Smith
Smith is not in list.
```

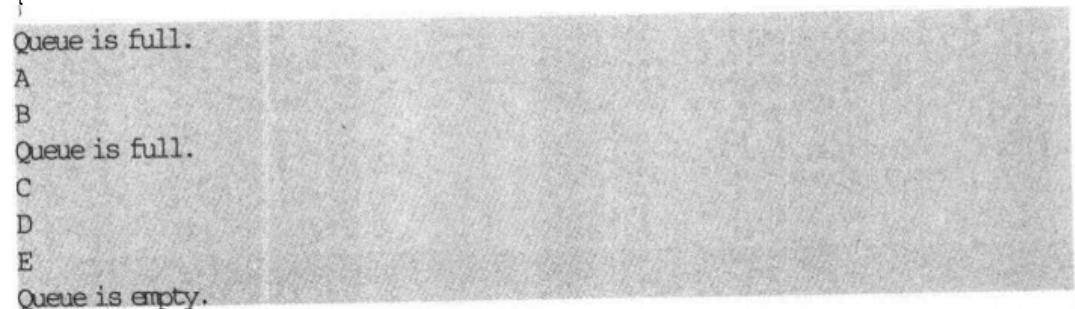
- 13.3 如 `Stack` 模板的实现那样，这个实现使用了一个类型为 `T` 的 `size` 个元素的序列 `data`。

数组中下个对象要插入的位置总是由值 ( $\text{front} \% \text{size}$ ) 给出, 保留下次要删除的对象的数组中的位置总是由值 ( $\text{rear} \% \text{size}$ ) 给出:

```
template< class T>
class Queue
{ public:
    Queue (int s = 100) : size (s+1), front (0), rear (0)
    { data = new T [size]; }
    ~Queue () { delete [] data; }
    void insert (const T& x) { data [rear++ % size] = x; }
    T remove () { return data [front++ % size]; }
    int isEmpty () const { return front == rear; }
    int isFull () const { return (rear+1) % size == front; }
private:
    int size, front, rear;
    T* data;
};
```

测试驱动程序使用了一个至多有 3 个字符的队列:

```
# include "Queue.h"
int main ()
{ Queue< char> q (3);
  q.insert ('A');
  q.insert ('B');
  q.insert ('C');
  if (q.isFull ()) cout << "Queue is full. \n";
  else cout << "Queue is not full. \n";
  cout << q.remove () << endl;
  cout << q.remove () << endl;
  q.insert ('D');
  q.insert ('E');
  if (q.isEmpty ()) cout << "Queue is empty. \n";
  else cout << "Queue is not empty. \n";
  cout << q.remove () << endl;
  cout << q.remove () << endl;
  cout << q.remove () << endl;
  if (q.isEmpty ()) cout << "Queue is empty. \n";
  else cout << "Queue is not empty. \n";
}
```



```
Queue is full.
A
B
Queue is full.
C
D
E
Queue is empty.
```

#### 13.4 这里增加了两个函数:

```
unsigned resize (unsigned n);
```

```
unsigned resize (unsigned n, T t);
```

两个函数都把向量转换成大小为  $n$  的向量。如果  $n < \text{size}$ ，那么最后  $\text{size} - n$  个元素简单地被去掉。如果  $n == \text{size}$ ，那么向量不变化。如果  $n > \text{size}$ ，那么要转换向量开始的  $\text{size}$  个元素应该和前面的版本相同；后面的  $n - \text{size}$  个元素被第二个函数 `resize()` 赋值为  $t$ ，并且没有被第一个函数初始化。两个函数都返回新的 `size`：

```
template < class T >
unsigned Vector< T >::resize (unsigned n, T t)
{ T* new_data = new T [n];
  copy (v);
  for (i = size; i < n; i++)
    new_data [i] = t;
  delete [] data;
  size = n;
  data = new_data;
  return size;
}

template< class T >
unsigned Vector< T >::resize (unsigned n)
{ T* new_data = new T [n];
  copy (v);
  delete [] data;
  size = n;
  data = new_data;
  return size;
}
```

### 13.5 新的构造函数转化一个元素类型为 $T$ 的数组 $a$ ：

```
template< class T >
class Vector
{ public:
  Vector (T* a) : size (sizeof (a)), data (new T [_size])
  { for (int i = 0; i < size; i++) data [i] = a [i];
    //其他的成员
  }
}
```

下面是一个新构造函数的测试驱动程序：

```
int main ()
{ int a [] = { 22, 44, 66, 88 };
  Vector< int > v (a);
  cout << v.size () << endl;
  for (int i = 0; i < 4; i++)
    cout << v [i] << " ";
}
```

4  
22 44 66 88

这个构造函数的好处是，现在可以初始化一个向量而不必给每个元素单独赋值。

### 13.6 派生的类模板有三个成员函数：两个构造函数和一个新的下标运算符：

```

template< class T, class E>
class Array : public Vector< T>
{
public:
    Array (E last) : Vector< T> (unsigned (last) + 1) {}
    Array (const Array< T, E> &a) : Vector< T> (a) {}
    T& operator [] (E index) const
    { return Vector< T> :: operator [] (unsigned (index)); }
};

```

第一个构造函数调用定义在父类 `Vector< T>` 中的默认的构造函数，传给它用做下标的 `E` 值的数目。新的复制构造函数和下标运算符也调用在它们父类中的等价函数。

下面是 `Array< T, E>` 模板的一个测试驱动程序：

```

enum Days { SUN, MON, TUE, WED, THU, FRI, SAT };

int main ()
{
    Array< int, Days> customers (SAT);
    customers [MON] = 27;
    customers [TUE] = 23;
    customers [WED] = 20;
    customers [THU] = 23;
    customers [FRI] = 36;
    customers [SAT] = customers [SUN] = 0;
    for (Days day = SUN; day != SAT; day++)
        cout << customers [day] << " ";
}

```

0 27 23 20 23 36 0

枚举类型 `Days` 为这个类型定义了 7 个值。然后对象 `customers` 被声明为一个由这 7 个值下标的整型数组。程序的剩余部分应用下标运算符来初始化，然后打印这个数组。

## 第 14 章 标准 C++ 向量

### 14.1 引言

尽管不是很有效率，标准 C++ 字符串对象比经典的 C 字符串更健壮。它们很容易使用并且能产生较少的运行时错误。同样，标准 C++ 向量对象比普通的数组更健壮。因此向量对象为数组提供了一个很好的替代。在标准 C++ 库中（见第 15 章），向量类模板也是所有的容器类的原型。

向量类模板在 `<vector>` 头文件中定义。

#### 例 14.1 使用一个字符串向量

这个程序生成一个有 8 个字符串的向量 `v`，然后调用一个 `load()` 函数和一个 `print()` 函数来装载和打印向量。

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
void load (vector< string> &);
void print (vector< string> );
const int SIZE=8;

int main ()
{ vector< string> v (SIZE);
  load (v);
  print (v);
}

void load (vector< string> &v)
{ v[0] = "Japan";
  v[1] = "Italy";
  v[2] = "Spain";
  v[3] = "Egypt";
  v[4] = "Chile";
  v[5] = "Zaire";
  v[6] = "Nepal";
  v[7] = "Kenya";
}

void print (vector< string> v)
```



```
for (int i = 0; i < SIZE; i++)
    cout << v[i] << endl;
cout << endl;
```

```
Japan
Italy
Spain
Egypt
Chile
Zaire
Nepal
Kenya
```

注意这个程序应和使用一个字符串数组的作用一样：

```
string v [SIZE];
```

值得一提的是，用下标运算符 `v[i]` 方式访问的作用和用向量和数组访问的作用一样。

### 例 14.2 使用 `push_back()` 和 `size()` 函数

这个程序和例 14.1 中的一样，除了用黑体字改变的部分：类型标识符 `strings` 用来替代 `vector<string>`，`push_back()` 函数用来替代把元素赋值给 `v[i]`，`size()` 函数用来替代把常量 `SIZE` 存储为一个全局常量。

```
typedef vector<string> strings;
void load (Strings&);
void print (Strings);

int main ()
{ Strings v;
  load (v);
  print (v);
}

void load (Strings& v)
{ v.push_back ("Japan");
  v.push_back ("Italy");
  v.push_back ("Spain");
  v.push_back ("Egypt");
  v.push_back ("Chile");
  v.push_back ("Zaire");
  v.push_back ("Nepal");
  v.push_back ("Kenya");
}

void print (strings v)
{ for (int i = 0; i < v.size (); i++)
    cout << v[i] << endl;
  cout << endl;
}
```

注意，向量 `v` 生成时有 0 个元素。每次 `push_back()` 被调用时，它把新元素加到向量

的末尾，增加了它的大小。因此，当 `load()` 函数返回时，`v` 的大小是 8。

这里的结果和例 14.1 中的一样。

## 14.2 关于向量的迭代符

### 例 14.3 使用向量迭代符

这个程序定义了类型标识符 `Sit` 来代表关于字符串向量的迭代符。然后，它使用这样的迭代符来遍历 `print()` 函数中的向量。

```
typedef vector<string> Strings;
typedef strings:: iterator Sit;
void load (Strings&);
void print (Strings);

int main ()
{ Strings v;
  load (v);
  print (v);
}

void print (Strings v)
{ for (sit it=v.begin (); it!=v.end (); it++)
  cout << *it << endl;
  cout << endl;
}
```

`for` 循环把迭代符 `it` 初始化为向量 `v` 的开始。表达式 `*it` 返回被迭代符定位的元素。增加表达式推进 `it` 到向量中的下一个元素。当 `it == v.end()` 时，它已经移到了紧随向量的最后一元素的想像中的位置。这标志着遍历已经结束，并停止了循环。

这里的输出和例 14.1 的程序相同。

### 例 14.4 使用一般的 `sort()` 算法

这儿使用定义在头文件 `<algorithm>` 中的 `sort()` 函数。随后的调用 `print(v)` 显示了串是按字母的顺序排序的。

```
int main ()
{ Strings v;
  load (v);
  sort (v.begin (), v.end ());
  print (v);
}
```

```
Chile
Egypt
Italy
Japan
```

```
Kenya
Nepal
Spain
Zaire
```

一般的 `sort()` 算法需要两个迭代符参数来表明向量的哪部分要排序。`begin()` 和 `end()` 函数返回定位向量的开始位置和结束位置的迭代符，因此，把两个迭代符传给 `sort()` 表明全部的向量要被排序。

### 14.3 赋值向量

#### 例 14.5 使用赋值运算符来复制一个向量

这个程序显示了一个向量可以被赋给另一个。

```
int main ()
{ Strings v, w;
  load (v);
  w = v;
  sort (v.begin (), v.end ());
  print (v);
  print (w);
}
```

```
Chile
Egypt
Italy
Japan
Kenya
Nepal
Spain
Zaire

Japan
Italy
Spain
Egypt
Chile
Zaire
Nepal
Kenya
```

赋值 `w = v` 和调用 `load (w)` 具有相同的效果：它复制 `v` 的 8 个元素中的每一个，并把它们装载入 `w`。

从输出来看，很明显有这个事实，`w` 独立于 `v`：即当 `v` 排序时，`w` 保持不变。

#### 例 14.6 使用 `front()`、`back()` 和 `pop_back()` 函数

`front()` 函数返回向量中的第一个元素。`back()` 函数返回向量中的最后一个元素。

`pop_back()` 函数删除向量中的最后一个元素。

```
int main ()
{ strings v;
  load (v);
  sort (v.begin (), v.end ());
  print (v);
  cout << "v.front () = " << v.front () << endl;
  cout << "v.back ()   = " << v.back () << endl;
  v.pop_back ();
  cout << "v.back ()   = " << v.back () << endl;
  v.pop_back ();
  cout << "v.back ()   = " << v.back () << endl;
  print (v);
}
```

调用 `v.pop_back()` 从向量 `v` 中去掉了字符串 `Zaire`。

```
Chile
Egypt
Italy
Japan
Kenya
Nepal
Spain
Zaire

v.front () = Chile
v.back ()   = Zaire
v.back ()   = Spain
v.back ()   = Nepal
Chile
Egypt
Italy
Japan
Kenya
Nepal
```

## 14.4 `erase()` 和 `insert()` 函数

### 例 14.7 使用 `erase()` 函数

```
int main ()
{ Strings v;
  load (v);
  sort (v.begin (), v.end ());
  print (v);
  v.erase (v.begin () + 2); //删除 Italy
  v.erase (v.end () - 2);   //删除 Spain
  print (v);
}
```

```

Chile
Egypt
Italy
Japan
Kenya
Nepal
Spain
Zaire

Chile
Egypt
Japan
Kenya
Nepal
Zaire

```

调用 `v.erase(v.begin() + 2)` 删除了元素 `v[2]`。它是紧随在从向量的开始的第二个元素 (Egypt)。

调用 `v.erase(v.begin() - 2)` 删除了元素 `v[n - 2]`。它是紧随在从向量的末尾的第二个元素 (Nepal)。

#### 例 14.8 使用 `insert()` 函数

这个程序说明了 `insert()` 函数，并且显示了使用 `erase()` 函数删除一整段的元素。

```

int main()
{
    strings v;
    load(v);
    sort(v.begin(), v.end());
    print(v);
    v.erase(v.begin() + 2, v.end() - 2); // 删除了从 Italy 到 Nepal 的段
    print(v);
    v.insert(v.begin() + 2, "India");
    print(v);
}

```

```

Chile
Egypt
Italy
Japan
Kenya
Nepal
Spain
Zaire

Chile
Egypt
Spain
Zaire

```

```
Chile
Egypt
India
Spain
Zaire
```

调用 `v.erase (v.begin () + 2, v.end () - 2)` 删除了段 `v [2..5]`

调用 `v.insert (v.begin () + 2, "India")` 把 India 立即插入到从向量开始的第二个元素 (Egypt) 后。

## 14.5 find()函数

`find ()` 函数用来查找向量中的一个元素

### 例 14.9 使用 `find ()` 函数

程序使用 `find ()` 函数来得到在向量中定位 Egypt 和 Malta 的迭代符。然后它把迭代符传递到 `sort ()` 函数来排列向量中的那个段的顺序

```
int main ()
{
    Strings v;
    load ();
    print (v);
    Sit egypt = find (v.begin (), v.end (), "Egypt");
    Sit malta = find (v.begin (), v.end (), "Malta");
    sort (egypt, malta);
    print (v);
}
```

```
void load (Strings &v)
{
    v.push_back ("Japan");
    v.push_back ("Italy");
    v.push_back ("Spain");
    v.push_back ("Egypt");
    v.push_back ("Chile");
    v.push_back ("Zaire");
    v.push_back ("Nepal");
    v.push_back ("Zer'ya");
    v.push_back ("India");
    v.push_back ("China");
    v.push_back ("Malta");
    v.push_back ("Syria");
}
```

```
Japan
Italy
Spain
Egypt
Chile
Zaire
Nepal
```

```
Kenya
India
China
Malta
Syria

Japan
Italy
Spain
Chile
China
Egypt
India
Kenya
Nepal
Zaire
Malta
Syria
```

两个迭代符 `egypt` 和 `malta` 由 `find()` 函数初始化。它们也描绘出段 `v[3..9]` 的 7 个元素 {Egypt, Chile, Zaire, Nepal, Kenya, India, China} 组成。`sort()` 函数排序这个段的。其他的 5 个元素没有变化。

如 `sort()` 函数, `find()` 函数是个一般的算法, 它要求两个迭代符指定要处理的向量的段。如果想查找全部的向量, 可使用由 `begin()` 函数和 `end()` 函数返回的迭代符, 例如:

```
find(v.begin(), v.end(), x);
```

## 14.6 C++ 标准向量类模板

向量类模板的接口就是所有标准 C++ 容器类模板的原型 (见第 15 章)。除了一些例外, 向量类的每个成员函数对应于每个其他容器类 (`stack`, `queue`, `list`, `set`, `map` 等) 的等价的成员函数。

下面是一个简化的 `vector` 类模板接口的部分列表:

```
template< class T>
class Vector
{
    friend bool operator == (const vector&, const vector&)
    friend bool operator < (const vector&, const vector&)
public:
    typedef T* iterator
    vector(); // 默认的构造函数
    vector(const vector&); // 复制的构造函数
    vector(int, const T&); // 辅助的构造函数
    vector(iterator, iterator); // 辅助的构造函数
    ~vector(); // 析构函数
    vector& operator = (const vector&); // 赋值运算符
```

```

void assign (int, const T&);           //赋一个给定的值
void assign (iterator, iterator);     //从对象复制元素
void resize (int);                    //改变向量的大小
void swap (vector&);                  //用对象交换元素
bool empty () const;                 //当且仅当是空时, 返回 true
int size () const;                    //返回元素的数目
iterator begin ();                    //定位第一个元素
iterator end ();                      //定位末尾的下一个元素
T& operator [] (int);                 //下标运算符
T& at (int);                          //范围测试访问
T& front ();                          //访问第一个元素
T& back ();                           //访问最后一个元素
void push_back (const T&);            //在末尾插入元素
void pop_back ();                     //删除最后一个元素
iterator insert (iterator, const T&);
void insert (iterator, int, const T&);
void insert (iterator, iterator, iterator);
iterator erase (iterator);
iterator erase (iterator, iterator);
void clear ();                        //删除所有的元素

private:
    //...
};

```

#### 例 14.10 使用标准 `vector< >` 类模板

下面是一个完整的 C++ 程序, 它使用了标准的 `vector< >` 类模板:

```

#include <iostream>
#include <vector>           // 定义标准的 vector< T> 类模板
using namespace std;
typedef vector< double> Vec;
typedef vector< bool> Bits;

template <class T>
void copy (vector< T> &v, const T* x, int n)
{ vector< T> w;
  for (int i = 0; i < n; i++)
    w.push_back (x[i]);
  v = w;
}

Vec projection (Vec& v, Bits& b)
{ int v_size = v.size ();
  assert (b.size () >= v_size);
  Vec w;
  for (int i = 0; i < v_size; i++)
    if (b[i]) w.push_back (v[i]);
  return w;
}

void print (Vec& v)

```



```

int v_size = v.size();
for (int i=0; i < v_size; i++)
    cout << v[i] << " ";
cout << endl;

int main ()
{
    double x[8] = { 22.2, 33.3, 44.4, 55.5, 66.6, 77.7, 88.8, 99.9 };
    Vec v;
    copy (x, x+8);
    bool y[8] = { false, true, false, true, true, true, false, true };
    Bits b;
    copy (b, y, 8);
    Vec w = projection (v, b);
    print (v);
    print (w);
}
22.2 33.3 44.4 55.5 66.6 77.7 88.8 99.9
33.3 55.5 66.6 77.7 99.9

```

它解释了向量类的 `push_back()` 和 `size()` 成员函数。

`projection(v, b)` 函数的目的是，使用位向量 `b` 作为一个删除向量 `v` 的选定元素的一个掩码。最终的向量 `w` 叫做由 `b` 决定的子空间上的投影。

## 14.7 范围检查

标准向量类模板的 `at()` 成员函数自动检测下标变量的值来确保它没有超出范围。这种预防程序失败的保护措施对普通的数组是不可能的。

## 复 习 题

- 14.1 数组和 C++ 向量的主要区别是什么？
- 14.2 向量迭代符和数组的下标如何相像？

## 习 题

- 14.1 使用 `find()` 算法实现并测试下面的整型向量的函数：

```

int frequency (vector<int> &v, int x);
//返回 v 中 x 出现的数目

```

- 14.2 使用 `find()` 算法和 `erase()` 函数来实现并测试下面的整型向量的函数：

```

void remove_duplicates (vector<int> &v);
//返回 v 中所有的复制品

```

- 14.3 使用 `sort()` 算法来实现并测试下面的浮点型向量的函数：

```
float median (vector< float> & v);
//返回 v 中排序元素中的中间数
```

#### 14.4 实现并测试下面的转化函数:

```
int unsignedValue (BinaryCode bc);
//例子: 如果 bc 的位值是 10111
//unsignedValue (bc) 返回 21
BinaryCode getUnsignedCode (unsigned n);
//返回 n 的最短的可能代码
//例子: 如果 n = 15, 用元素 1111 返回向量
int signedValue (BinaryCode bc);
//例子: 如果 bc 的位值是 101110
//signedValue (bc) 返回 -30
BinaryCode getSignedCode (int n);
//返回 n 的最短的可能的二进制完整代码
//例子: 如果 n = 15, 用元素 01111 返回向量
//如果 n = -15, 用元素 10001 返回向量
```

这些使用了下面的定义:

```
typedef vector< int> BinaryCode;
typedef BinaryCode:: iterator BCIterator;
```

## 复习题答案

#### 14.1 数组和 C++ 向量的-一些主要区别是:

a. 一个数组声明为:

```
string s[8]; a; // a 是一个有 8 个字符串的数组
而一个向量声明为:
vector< String> v (8); // v 是一个有 8 个字符串的向量
```

b. 赋值运算符是为向量而不是为数组定义的:

```
v = w; //把向量 w 的所有元素赋给 v
```

c. 比较运算符是为向量而不是为数组定义的:

```
v == w //如果两个向量相等返回 true
v < w //使用按词典的顺序的向量
```

d. size () 函数只有向量有而数组没有:

```
int n = v.size (); //向量中的元素数目
```

e. at () 函数只有向量有而数组没有:

```
string s8 = v.at (8); //位置为 8 的元素
```

如果元素不存在, 就会出现一个超出范围的错误例程

#### 14.2 数组索引和向量迭代符的一些主要相似点是:

a. 两个都提供对元素的直接读-写访问:

```

x = a[3];    //赋给 x 第 3 个元素
x = *it;     //赋给 x 的是由 it 定位的元素
a[3] = 44;   //把 44 赋给第 3 个元素
*it = 44;    //把 44 赋给由 it 定位的元素

```

b. 两个都可以增加和减少。

c. 两个都可以作为相关位置的要素：

```

x = a[i+3];  //赋给 x a[i] 后的第三个元素
x = *(it+3); //赋给 x *it 后的第三个元素

```

## 习 题 答 案

```

14.1 int frequency (vector<int> v, int x)
{ int n=0;
  for (vector<int>::iterator it=v.begin(); it!=v.end(); it++)
    if (*it==x) n++;
  return n;
}

14.2 void remove_duplicates (vector<int> &v)
{ for (vector<int>::iterator it=v.begin(); it!=v.end(); it++)
    if (it!=v.begin() && *it==*(it-1))
        it=v.erase(it);
}

14.3 typedef vector<float> ScoreVector;
typedef ScoreVector::iterator ScoreVectorIterator;

float median (ScoreVector sv):
//前提条件: sv 不空
//返回 sv 中两个排序的中间值的平均值
//调用者的参数没有变化

void getScores (ScoreVector &sv);
void print (ScoreVectorIterator start, ScoreVectorIterator stop);

int main ()
{
    ScoreVector scores;
    getScores (scores);
    print (scores.begin(), scores.end() - 1);
    cout << "median ( scores ) = " << median (scores) << endl;
}

float median (ScoreVector v)
{

```

```

        if (v.empty ()) return 0.0;
        int n = v.size ();
        sort ( v.begin (), v.end () );
        return (v [n/2] + v [ (n - 1) / 2 ]) / 2.0;
    }

    void getScores (ScoreVector & sv)
    {
        float nextScore;
        cout << "Enter next score or negative value to stop: ";
        cin >> nextScore;
        while (nextScore >= 0.0)
        { sv.push_back (nextScore);
          cout << "Enter next score or negative value to stop: ";
          cin >> nextScore;
        }
    }

    void print (ScoreVectorIterator start, ScoreVectorIterator stop)
    {
        for (ScoreVectorIterator svIt = start; svIt <= stop; svIt++)
            cout << *svIt << endl;
    }

```

14.4 typedef vector< int > BinaryCode ;  
 typedef BinaryCode:: iterator BCIterator ;

```

int unsignedValue ( BinaryCode bc );
//例子: 如果 bc 的位值是 10101
//unsignedValue (bc) 返回 21

```

```

BinaryCode getUnsignedCode ( unsigned n );
//返回 n 的最短的可能代码
//例子: 如果 n = 15, 用元素 1111 返回向量

```

```

int signedValue ( BinaryCode bc );
//例子: 如果 bc 的位值是 101110
//signedValue (bc) 返回 -30

```

```

BinaryCode getSignedCode ( int n );
//返回 n 的最短的可能二进制完整代码
//例子: 如果 n = 15, 用元素 01111 返回向量
        如果 n = -15, 用元素 10001 返回向量

```

```

void print ( BinaryCode bc );
void testUnsigned ();
void testSigned ();

```

```

int main ()
{ testUnsigned ();
  testSigned ();
}

```

```

void testUnsigned ()
{
    BinaryCode bc ;
    for ( unsigned n = 0 ; n <= 11 ; n++ )
    {
        bc = getUnsignedCode ( n ) ;
        print ( bc ) ;
        cout << " has unsigned value" << unsignedValue ( bc )
            << " and signed value" << signedValue ( bc ) << endl ;
    }
}

int unsignedValue ( BinaryCode bc )
{
    int value = 0 ;
    for ( BCIterator bcIt = bc.begin () ; bcIt != bc.end () ; bcIt++ )
        value = value * 2 + *bcIt ;
    return value ;
}

BinaryCode getUnsignedCode ( unsigned n )
{
    BinaryCode answer ;
    answer.push_back ( n % 2 ) ; // 以最少的数字位开始
    n = n / 2 ;
    while ( n > 0 )
    {
        BCIterator bcIt = answer.begin () ;
        answer.insert ( bcIt , n % 2 ) ;
        n = n / 2 ;
    }
    return answer ;
}

void print ( BinaryCode bc )
{
    for ( BCIterator bcIt = bc.begin () ; bcIt != bc.end () ; bcIt++ )
        cout << *bcIt << ' ' ;
}

int signedValue ( BinaryCode bc )
{
    int uvalue = unsignedValue ( bc ) ;
    if ( *bc.begin () == 0 ) return uvalue ; // 非负
    int modulus = (int) pow ( 2 , bc.size () ) ;
    return uvalue - modulus ;
}

BinaryCode getSingedCode ( int n )
{
    BinaryCode answer ;
    if ( n >= 0 ) // n 非负
    {
        answer = getUnsignedCode ( n ) ;
        BCIterator bcIt = answer.begin () ;
        answer.insert ( bcIt , 0 ) ; // 插入主要的位 0
    }
    else // n 负
    {
        int posN = -n ;
        int modulus = 2 ;
        while ( posN > 0 ) // 设定 modulus
            posN /= 2 ;
    }
}

```

```

        modulus * = 2 ;
    }
    answer = getUnsignedCode ( modulus + n ) ;
}
return answer ;
}

void testSigned ()
{
    BinaryCode bcPos ;
    BinaryCode bcNeg ;
    for ( int n = 1 ; n <= 12 ; n++ )
    {
        bcPos = getSignedCode ( n ) ;
        bcNeg = getSignedCode ( -n ) ;
        int decodePos = signedValue ( bcPos ) ;
        int decodeNeg = signedValue ( bcNeg ) ;
        cout << decodePos << " : " ;
        print ( bcPos ) ;
        cout << "\tvs\t\t" << decodeNeg << " : " ;
        print ( bcNeg ) ;
        cout << endl ;
    }
}

```

# 第 15 章 容 器 类

## 15.1 ANSI/ISO 标准 C++

ANSI（美国国家标准化组织）和 ISO（国际标准化组织）的 C++ 标准化开始于 1989 年。最终的版本也是由这两个组织在 1998 正式批准的。那次批准定义了标准的 C++。

可以在网址：

<http://www.ansi.org>

从 ANSI 得到一个标准的复制品，这个文档的标题是 Information Technology—Programming Languages—C++。

## 15.2 标准模板库

C++ 的标准化带来了许多变化，包括命名空间和一个正式的 bool 类型。但最大的改进就是增加了标准模板库（STL）。这是一个类模板和函数的集合，它有助于容器对象的使用，这些对象如字符串、向量、列表、栈、队列、集合、图等。STL 是在 Hewlett—Packard 由 Alexander Stepanov 领导的一个小组设计的，STL 现在只是作为标准 C++ 库的一部分而闻名。从这些模板可以定义的类型叫做容器类。

## 15.3 标准 C++ 容器类模板

这十个标准 C++ 容器类模板组织如下（附录 C 中给出了这些类模板的详细情况）：

Container（容器）是一个包含其他对象的数据结构。它包含的对象叫做元素。一个给定的容器中的所有元素必须有相同的类型。

Sequence container（序列容器）是一个元素在一个顺序的序列中保存的容器，如数组。每个元素的位置和它的值是独立的。但除非是故意移动元素，它们的相对位置应确保不变。就像图 15.1 所示的那样，共有三个一般的序列容器：vector、deque 和 list。

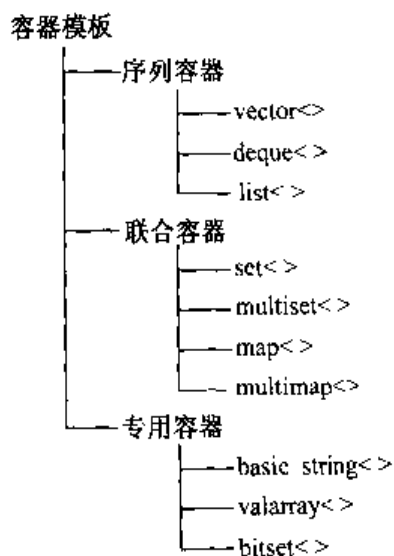


图 15.1 容器模板

**associative container** (联合容器) 是一个元素以已排定的顺序保存的元素。因此用户不能控制元素存放在何处; 在容器中它们的位置完全由它们的值和其他元素的值决定。因此插入元素的顺序并不重要。就像图 15.1 中所示的那样, 共有四个一般的 **associative container**: **set**、**multiset**、**map** 和 **multimap**。

标准 C++ 库也定义了三个特殊的容器模板: **basic\_string**、**valarray** 和 **bitset**。没有把它们归类为一般的容器类是因为它们的运算符和其他的容器类不一样。

**vector** < > 模板是所有容器类的原型。就像第 10 章中描述的那样, 它归纳出直接访问的数组。它的大部分函数适用于其他的模板。

**vector** < > 模板在第 14 章中有描述。

**deque** < > 模板归纳了栈容器和队列容器。**deque** (发音 “deck”) 是一个允许在末端进行插入和删除的序列容器。标准 C++ 库提供了专门的适配器来使用这个模板定义 **stack** < > 模板和 **queue** < > 模板。

**list** < > 模板归纳了链表结构, 它没有直接的索引访问, 但它的确有更快的插入和删除操作。一个专门的适配器使用 **list** < > 模板来定义 **priority\_queue** < > 模板。

**set** < > 模板提供了表示数学上的集合、使用合并和并集运算的容器。

**multiset** < > 模板和 **set** < > 模板相同, 除了它的容器允许多个复制的元素外。

**map** < > 模板归纳了查询表结构。映射也叫做一个关联数组。哈希表数据结构就是一个特殊种类的映射。

**multimap** < > 模板和 **map** < > 模板相同, 除了它的容器允许多个复制的元素外。

**basic\_string** < > 模板归纳了字符串的定义, 它允许任何类型的字符串。一般的特殊例子由 **typedef** 定义:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

**valarray** < > 模板是为实例化数学上的向量和线性数组处理而设计的。

**bitset** < > 模板用来处理位串; 其值通常是十六进制的并且是由逻辑运算符 **|**、**&**、**<<** 和 **>>** 操作的对象。

## 15.4 标准 C++ 的一般算法

标准 C++ 一般的算法适用于标准 C++ 容器类的非成员函数。它们提供了一套连贯的工具, 差不多覆盖了容器的任何应用程序。它们也允许很简单地从一个容器类型到另一个容器类型的转化。这些函数的细节在附录 D 中给出。

**find()** 函数和 **sort()** 函数是两个最有用的算法。它们在第 14 章中有描述 (见例 14.4 和例 14.9), 在这一章的例子中它们是和其他的容器一起描述的。



## 15.5 头文件

标准 C++ 容器模板和一般的算法定义在下面的头文件中:

|                                     |                                  |
|-------------------------------------|----------------------------------|
| <code>accumulate ()</code>          | <code>&lt; numeric &gt;</code>   |
| <code>adjacent_difference ()</code> | <code>&lt; numeric &gt;</code>   |
| <code>adjacent_find ()</code>       | <code>&lt; algorithm &gt;</code> |
| <code>basic_string::&gt;</code>     | <code>&lt; string &gt;</code>    |
| <code>binary_search ()</code>       | <code>&lt; algorithm &gt;</code> |
| <code>bitset&lt; &gt;</code>        | <code>&lt; bitset &gt;</code>    |
| <code>copy ()</code>                | <code>&lt; algorithm &gt;</code> |
| <code>copy_backward ()</code>       | <code>&lt; algorithm &gt;</code> |
| <code>count ()</code>               | <code>&lt; algorithm &gt;</code> |
| <code>count_if ()</code>            | <code>&lt; algorithm &gt;</code> |
| <code>deque ()</code>               | <code>&lt; deque &gt;</code>     |
| <code>equal ()</code>               | <code>&lt; algorithm &gt;</code> |
| <code>equal_find ()</code>          | <code>&lt; algorithm &gt;</code> |
| <code>fill ()</code>                | <code>&lt; algorithm &gt;</code> |
| <code>fill_n ()</code>              | <code>&lt; algorithm &gt;</code> |
| <code>find_end ()</code>            | <code>&lt; algorithm &gt;</code> |
| <code>find_first_of ()</code>       | <code>&lt; algorithm &gt;</code> |
| <code>find_if ()</code>             | <code>&lt; algorithm &gt;</code> |
| <code>for_each ()</code>            | <code>&lt; algorithm &gt;</code> |
| <code>generate ()</code>            | <code>&lt; algorithm &gt;</code> |
| <code>generate_n ()</code>          | <code>&lt; algorithm &gt;</code> |
| <code>includes ()</code>            | <code>&lt; algorithm &gt;</code> |
| <code>inner_product ()</code>       | <code>&lt; algorithm &gt;</code> |
| <code>inplace_merge ()</code>       | <code>&lt; algorithm &gt;</code> |
| <code>iter_swap ()</code>           | <code>&lt; algorithm &gt;</code> |
| <code>lexicographic_compare</code>  | <code>&lt; algorithm &gt;</code> |
| <code>list&lt; &gt;</code>          | <code>&lt; list &gt;</code>      |
| <code>lower_bound ()</code>         | <code>&lt; algorithm &gt;</code> |
| <code>make_heap ()</code>           | <code>&lt; algorithm &gt;</code> |
| <code>map ()</code>                 | <code>&lt; map &gt;</code>       |
| <code>max ()</code>                 | <code>&lt; algorithm &gt;</code> |
| <code>max_element ()</code>         | <code>&lt; algorithm &gt;</code> |
| <code>merge ()</code>               | <code>&lt; algorithm &gt;</code> |
| <code>min ()</code>                 | <code>&lt; algorithm &gt;</code> |
| <code>min_element ()</code>         | <code>&lt; algorithm &gt;</code> |
| <code>mismatch ()</code>            | <code>&lt; algorithm &gt;</code> |
| <code>multimap&lt; &gt;</code>      | <code>&lt; map &gt;</code>       |
| <code>multiset&lt; &gt;</code>      | <code>&lt; set &gt;</code>       |
| <code>next_permutation ()</code>    | <code>&lt; algorithm &gt;</code> |
| <code>nth_element ()</code>         | <code>&lt; algorithm &gt;</code> |
| <code>partial_sort ()</code>        | <code>&lt; algorithm &gt;</code> |
| <code>partial_sum ()</code>         | <code>&lt; numeric &gt;</code>   |
| <code>partition ()</code>           | <code>&lt; algorithm &gt;</code> |
| <code>partition_sort_copy ()</code> | <code>&lt; algorithm &gt;</code> |
| <code>pop_heap ()</code>            | <code>&lt; algorithm &gt;</code> |
| <code>prev_permutation ()</code>    | <code>&lt; algorithm &gt;</code> |

|                            |               |
|----------------------------|---------------|
| priority_queue()           | < queue >     |
| push_heap()                | < algorithm > |
| queue()                    | < queue >     |
| random_shuffle()           | < algorithm > |
| remove_copy()              | < algorithm > |
| remove_copy_if()           | < algorithm > |
| remove_if()                | < algorithm > |
| replace()                  | < algorithm > |
| replace_copy()             | < algorithm > |
| replace_copy_if()          | < algorithm > |
| replace_if()               | < algorithm > |
| reverse()                  | < algorithm > |
| reverse_copy()             | < algorithm > |
| rotate()                   | < algorithm > |
| rotate_copy()              | < algorithm > |
| search_n()                 | < algorithm > |
| set()                      | < set >       |
| set_difference()           | < algorithm > |
| set_intersection()         | < algorithm > |
| set_symmetric_difference() | < algorithm > |
| set_union()                | < algorithm > |
| sort()                     | < algorithm > |
| sort_heap()                | < algorithm > |
| stack< >                   | < stack >     |
| string< >                  | < vector >    |
| swap()                     | < algorithm > |
| transform()                | < algorithm > |
| unique()                   | < algorithm > |
| unique_copy()              | < algorithm > |
| upper_bound()              | < algorithm > |
| valarray< >                | < valarray >  |
| vector< >                  | < vector >    |

关于标准 C++ 容器类和它们的一般算法的更多信息, 可以查看附录 H 中列出的书 [Hubbard1] 和 [Hubbard2]。

# 附录 A 字符代码

## A.1 ASCII 码

每个 8 位的字符串都是以它的 ASCII<sup>①</sup>码存储的，这是一个在范围 0 ~ 127 内的整数。注意，前 32 个字符是“显示非打印字符”，因此，在第一列中它们的符号或者用它们的 control sequence，或者用它们的 escape sequence 表示。“显示非打印字符”的 Control sequence 是 Control 和另一个从键盘输入的字符的组合。例如，“文件结束”字符（ASCII 码第 4 个）是用 Ctrl - D sequence 输入的。“显示非打印字符”的 escape sequence 是反斜线符号“\”（叫做“控制字符”）和一个在 C++ 源代码中表示字符的字母的组合。例如，“换行”字符（ASCII 码第 10 个）在一个 C++ 程序中写作“\n”。

| 字符       | 描述          | 十进制 | 八进制 | 十六进制 | 二进制   |
|----------|-------------|-----|-----|------|-------|
| Ctrl - @ | 空，串结束       | 0   | 000 | 0x0  | 0     |
| Ctrl - A | 标题开始        | 1   | 001 | 0x1  | 1     |
| Ctrl - B | 文本开始        | 2   | 002 | 0x2  | 10    |
| Ctrl - C | 文本结束        | 3   | 003 | 0x3  | 11    |
| Ctrl - D | 传输结束，文件结束   | 4   | 004 | 0x4  | 100   |
| Ctrl - E | 询问          | 5   | 005 | 0x5  | 101   |
| Ctrl - F | 肯定应答        | 6   | 006 | 0x6  | 110   |
| \a       | 钟，警报，系统哔哔声  | 7   | 007 | 0x7  | 111   |
| \b       | 退后一格        | 8   | 010 | 0x8  | 1000  |
| \t       | 水平制表        | 9   | 011 | 0x9  | 1001  |
| \n       | 行满，换行       | 10  | 012 | 0xa  | 1010  |
| \v       | 竖直制表        | 11  | 013 | 0xb  | 1011  |
| \f       | 表格满，换页      | 12  | 014 | 0xc  | 1100  |
| \r       | 回车          | 13  | 015 | 0xd  | 1101  |
| Ctrl - N |             | 14  | 016 | 0xe  | 1110  |
| Ctrl - O |             | 15  | 017 | 0xf  | 1111  |
| Ctrl - P | 数据连接码       | 16  | 020 | 0x10 | 10000 |
| Ctrl - Q | 设备控制 1，恢复滚屏 | 17  | 021 | 0x11 | 10001 |
| Ctrl - R | 设备控制 2      | 18  | 022 | 0x12 | 10010 |
| Ctrl - S | 设备控制 3，停止滚屏 | 19  | 023 | 0x13 | 10011 |

① ASCII 是 America Standard Code for Information Interchange 的只取首字母的缩写词

(续表)

| 字符       | 描述         | 十进制 | 八进制 | 十六进制 | 二进制    |
|----------|------------|-----|-----|------|--------|
| Ctrl - T | 设备控制 4     | 20  | 024 | 0x14 | 10100  |
| Ctrl - U | 拒绝应答       | 21  | 025 | 0x15 | 10101  |
| Ctrl - V | 同步停顿       | 22  | 026 | 0x16 | 10110  |
| Ctrl - W | 结束传输块      | 23  | 027 | 0x17 | 10111  |
| Ctrl - X | 取消         | 24  | 030 | 0x18 | 11000  |
| Ctrl - Y | 消息结束, 中断   | 25  | 031 | 0x19 | 11001  |
| Ctrl - Z | 替换, 退出     | 26  | 032 | 0x1a | 11010  |
| Ctrl - [ | 换码         | 27  | 033 | 0x1b | 11011  |
| Ctrl - / | 文件分隔符      | 28  | 034 | 0x1c | 11100  |
| Ctrl - ] | 群分隔符       | 29  | 035 | 0x1d | 11101  |
| Ctrl - ^ | 记录分隔符      | 30  | 036 | 0x1e | 11110  |
| Ctrl - _ | 单元分隔符      | 31  | 037 | 0x1f | 11111  |
|          | 空格         | 32  | 040 | 0x20 | 100000 |
| !        | 感叹号        | 33  | 041 | 0x21 | 100001 |
| "        | 引号标志, 双引号  | 34  | 042 | 0x22 | 100010 |
| #        | 散列符号, 数字符号 | 35  | 043 | 0x23 | 100011 |
| \$       | 美元符号       | 36  | 044 | 0x24 | 100100 |
| %        | 百分比号       | 37  | 045 | 0x25 | 100101 |
| &        | 表示 and 的符号 | 38  | 046 | 0x26 | 100110 |
| '        | 省略号, 单引号   | 39  | 047 | 0x27 | 100111 |
| (        | 左圆括号       | 40  | 050 | 0x28 | 101000 |
| )        | 右圆括号       | 41  | 051 | 0x29 | 101001 |
| *        | 星号, 星, 乘   | 42  | 052 | 0x2a | 101010 |
| +        | 加号         | 43  | 053 | 0x2b | 101011 |
| ,        | 逗号         | 44  | 054 | 0x2c | 101100 |
| -        | 破折号, 减号    | 45  | 055 | 0x2d | 101101 |
| .        | 点, 句点, 小数点 | 46  | 056 | 0x2e | 101110 |
| /        | 斜线         | 47  | 057 | 0x2f | 101111 |
| 0        | 阿拉伯数字 0    | 48  | 060 | 0x30 | 110000 |
| 1        | 阿拉伯数字 1    | 49  | 061 | 0x31 | 110001 |
| 2        | 阿拉伯数字 2    | 50  | 062 | 0x32 | 110010 |
| 3        | 阿拉伯数字 3    | 51  | 063 | 0x33 | 110011 |
| 4        | 阿拉伯数字 4    | 52  | 064 | 0x34 | 110100 |
| 5        | 阿拉伯数字 5    | 53  | 065 | 0x35 | 110101 |
| 6        | 阿拉伯数字 6    | 54  | 066 | 0x36 | 110110 |
| 7        | 阿拉伯数字 7    | 55  | 067 | 0x37 | 110111 |
| 8        | 阿拉伯数字 8    | 56  | 070 | 0x38 | 111000 |
| 9        | 阿拉伯数字 9    | 57  | 071 | 0x39 | 111001 |
| :        | 冒号         | 58  | 072 | 0x3a | 111010 |
| ;        | 分号         | 59  | 073 | 0x3b | 111011 |

(续表)

| 字符 | 描述     | 十进制 | 八进制  | 十六进制 | 二进制     |
|----|--------|-----|------|------|---------|
| <  | 小于     | 60  | 074  | 0x3c | 111100  |
| =  | 等于     | 61  | 075  | 0x3d | 111101  |
| >  | 大于     | 62  | 076  | 0x3e | 111110  |
| ?  | 问号     | 63  | 077  | 0x3f | 111111  |
| @  | 商业的标志  | 64  | 0100 | 0x40 | 1000000 |
| A  | 大写字母 A | 65  | 0101 | 0x41 | 1000001 |
| B  | 大写字母 B | 66  | 0102 | 0x42 | 1000010 |
| C  | 大写字母 C | 67  | 0103 | 0x43 | 1000011 |
| D  | 大写字母 D | 68  | 0104 | 0x44 | 1000100 |
| E  | 大写字母 E | 69  | 0105 | 0x45 | 1000101 |
| F  | 大写字母 F | 70  | 0106 | 0x46 | 1000110 |
| G  | 大写字母 G | 71  | 0107 | 0x47 | 1000111 |
| H  | 大写字母 H | 72  | 0110 | 0x48 | 1001000 |
| I  | 大写字母 I | 73  | 0111 | 0x49 | 1001001 |
| J  | 大写字母 J | 74  | 0112 | 0x4a | 1001010 |
| K  | 大写字母 K | 75  | 0113 | 0x4b | 1001011 |
| L  | 大写字母 L | 76  | 0114 | 0x4c | 1001100 |
| M  | 大写字母 M | 77  | 0115 | 0x4d | 1001101 |
| N  | 大写字母 N | 78  | 0116 | 0x4e | 1001110 |
| O  | 大写字母 O | 79  | 0117 | 0x4f | 1001111 |
| P  | 大写字母 P | 80  | 0120 | 0x50 | 1010000 |
| Q  | 大写字母 Q | 81  | 0121 | 0x51 | 1010001 |
| R  | 大写字母 R | 82  | 0122 | 0x52 | 1010010 |
| S  | 大写字母 S | 83  | 0123 | 0x53 | 1010011 |
| T  | 大写字母 T | 84  | 0124 | 0x54 | 1010100 |
| U  | 大写字母 U | 85  | 0125 | 0x55 | 1010101 |
| V  | 大写字母 V | 86  | 0126 | 0x56 | 1010110 |
| W  | 大写字母 W | 87  | 0127 | 0x57 | 1010111 |
| X  | 大写字母 X | 88  | 0130 | 0x58 | 1011000 |
| Y  | 大写字母 Y | 89  | 0131 | 0x59 | 1011001 |
| Z  | 大写字母 Z | 90  | 0132 | 0x5a | 1011010 |
| [  | 左括弧    | 91  | 0133 | 0x5b | 1011011 |
| \  | 反斜线    | 92  | 0134 | 0x5c | 1011100 |
| ]  | 右括弧    | 93  | 0135 | 0x5d | 1011101 |
| ^  | 脱字符号   | 94  | 0136 | 0x5e | 1011110 |
| _  | 下划线    | 95  | 0137 | 0x5f | 1011111 |
| ,  | 重音标记   | 96  | 0140 | 0x60 | 1100000 |
| a  | 小写字母 A | 97  | 0141 | 0x61 | 1100001 |
| b  | 小写字母 B | 98  | 0142 | 0x62 | 1100010 |
| c  | 小写字母 C | 99  | 0143 | 0x63 | 1100011 |

(续表)

| 字符     | 描述     | 十进制 | 八进制  | 十六进制 | 二进制     |
|--------|--------|-----|------|------|---------|
| d      | 小写字母 D | 100 | 0144 | 0x64 | 1100100 |
| e      | 小写字母 E | 101 | 0145 | 0x65 | 1100101 |
| f      | 小写字母 F | 102 | 0146 | 0x66 | 1100110 |
| g      | 小写字母 G | 103 | 0147 | 0x67 | 1100111 |
| h      | 小写字母 H | 104 | 0150 | 0x68 | 1101000 |
| i      | 小写字母 I | 105 | 0151 | 0x69 | 1101001 |
| j      | 小写字母 J | 106 | 0152 | 0x6a | 1101010 |
| k      | 小写字母 K | 107 | 0153 | 0x6b | 1101011 |
| l      | 小写字母 L | 108 | 0154 | 0x6c | 1101100 |
| m      | 小写字母 M | 109 | 0155 | 0x6d | 1101101 |
| n      | 小写字母 N | 110 | 0156 | 0x6e | 1101110 |
| o      | 小写字母 O | 111 | 0157 | 0x6f | 1101111 |
| p      | 小写字母 P | 112 | 0160 | 0x70 | 1110000 |
| q      | 小写字母 Q | 113 | 0161 | 0x71 | 1110001 |
| r      | 小写字母 R | 114 | 0162 | 0x72 | 1110010 |
| s      | 小写字母 S | 115 | 0163 | 0x73 | 1111100 |
| t      | 小写字母 T | 116 | 0164 | 0x74 | 1110100 |
| u      | 小写字母 U | 117 | 0165 | 0x75 | 1110101 |
| v      | 小写字母 V | 118 | 0166 | 0x76 | 1110110 |
| w      | 小写字母 W | 119 | 0167 | 0x77 | 1110111 |
| x      | 小写字母 X | 120 | 0170 | 0x78 | 1111000 |
| y      | 小写字母 Y | 121 | 0171 | 0x79 | 1111001 |
| z      | 小写字母 Z | 122 | 0172 | 0x7a | 1111010 |
| {      | 左大括弧   | 123 | 0173 | 0x7b | 1111011 |
|        | 管道     | 124 | 0174 | 0x7c | 1111100 |
| }      | 右大括弧   | 125 | 0175 | 0x7d | 1111101 |
| ~      | 标题     | 126 | 0176 | 0x7e | 1111110 |
| Delete | 删除、擦去  | 127 | 0177 | 0x7f | 1111111 |

## A.2 Unicode

Unicode 是 C++ 使用的它的 16 位 `wchar_t` (宽字符) 类型的国际化的字符集合。每个代码是一个带有惟一值的 16 位的整数, 值的范围是 0 到 65 535。这些值通常以十六进制的形式表达。(见附录 G) 例如, 无穷大符号的 Unicode 值是 8734, 它的十六进制形式是 `0x0000221e`。

在 C++ 中, Unicode 值以十六进制形式表达是 `0x0000hhhh` 的字符文字被表示为 `L' \xhhhh'`。例如, 无穷大符号表达为 `L' \x221e'`, 如下所示:

wchar\_t infinity = L'\x221e';

开始的 127 个 Unicode 值和 ASCII 码一样编码相同的字符。

下表总结了各种各样的字母和它们的 Unicode。

可以从 Unicode 协会的网址获得更多的信息：

<http://www.unicode.org/>

可以查看在附录 H 中列出的书 [Unicode]。

| Range (Hexadecimal): 范围 (十六进制) | Alphabets: 字母表                         |
|--------------------------------|----------------------------------------|
| \ u0000 - \ u024F              | Latin Alphabets: 拉丁字母                  |
| \ u0370 - \ u03FF              | Greek: 希腊语                             |
| \ u0400 - \ u04FF              | Cyrillic: 古斯拉夫语                        |
| \ u0530 - \ u058F              | Armenian: 亚美尼亚语                        |
| \ u0590 - \ u05FF              | Hebrew: 希伯来语                           |
| \ u0600 - \ u06FF              | Arabic: 阿拉伯语                           |
| \ u0900 - \ u097F              | Devanagari: 梵文字母                       |
| \ u0980 - \ u09FF              | Bengali: 孟加拉语                          |
| \ u0A00 - \ u0A7F              | Gurmukhi: 果鲁穆奇语                        |
| \ u0A80 - \ u0AFF              | Gujarati: 古吉拉特语                        |
| \ u0B00 - \ u0B7F              | Oriya 奥里雅语印度东部奥里雅州的一种印度语               |
| \ u0B80 - \ u0BFF              | Tamil: 泰米尔语                            |
| \ u0C00 - \ u0C7F              | Teluga:                                |
| \ u0C80 - \ u0CFF              | Kannada<印> 埃纳德语                        |
| \ u0D00 - \ u0D7F              | Malayam: 马来语                           |
| \ u0E00 - \ u0E7F              | Thai: 泰国语                              |
| \ u0E80 - \ u0EFF              | Lao: 老挝语                               |
| \ u0F00 - \ u0FBF              | Tibetan: 西藏语,                          |
| \ u10A0 - \ u10FF              | Georgian: 乔治亚语                         |
| \ u1100 - \ u11FF              | Hangul Jamo:                           |
| \ u2000 - \ u206F              | Punctuation: 标点符号                      |
| \ u2070 - \ u209F              | Superscripts and subscripts: 上标和下标     |
| \ u20A0 - \ u20CF              | Currency symbols: 货币符号                 |
| \ u20D0 - \ u20FF              | Diacritical marks: 可区分的标志              |
| \ u2100 - \ u214F              | Letterlike symbols: 像字母似的符号            |
| \ u2150 - \ u218F              | Numeral forms: 数字形式                    |
| \ u2190 - \ u21FF              | Arrows: 箭头                             |
| \ u2200 - \ u22FF              | Mathematical symbols                   |
| \ u2300 - \ u23FF              | Miscellaneous technical symbols:       |
| \ u2400 - \ u243F              | Control pictures:                      |
| \ u2440 - \ u245F              | Optical character Recognition symbols: |
| \ u2460 - \ u24FF              | Enclosed alphanumerics                 |
| \ u2500 - \ u257F              | Box drawing:                           |

(续表)

| Range (Hexadecimal): 范围 (十六进制) | Alphabets: 字母表                   |
|--------------------------------|----------------------------------|
| \ u2580 - \ u259F              | Block elements: 块元素              |
| \ u25A0 - \ u25FF              | Geometric shapes: 几何形状           |
| \ u2700 - \ u27BF              | Dingbats: 俚语                     |
| \ u3040 - \ u309F              | Hiragana: <日> 平假名                |
| \ u30A0 - \ u30FF              | Katakana: (日本字母) 片假名             |
| \ u3100 - \ u312F              | Bopomofo: 汉语拼音字母                 |
| \ u3130 - \ u318F              | Jamo                             |
| \ u3190 - \ u319F              | Kanbun                           |
| \ u3200 - \ u32FF              | Enclosed CJK letters and months: |
| \ u4E00 - \ u9FFF              | CJK Ideographs: CJK 表意文字         |



## 附录 B 标准 C++ 关键字

| 关键字          | 描述                     | 例子                          |
|--------------|------------------------|-----------------------------|
| and          | AND 运算符 && 的同义词        | (x > 0 and x < 8)           |
| and_eq       | 位运算 AND 赋值运算符 &= 的同义词  | b1 and_eq b2;               |
| asm          | 允许信息直接传递给汇编程序          | asm ("check");              |
| auto         | 只在它们自己的模块存在的对象存储类型     | auto int n;                 |
| bitand       | 位运算 AND 运算符 & 的同义词     | b0 = b1 bitand b2;          |
| bitor        | 位运算 OR 运算符   的同义词      | b0 = b1 bitor b2;           |
| bool         | 布尔类型                   | bool flag;                  |
| break        | 结束一个循环或 switch 语句      | break;                      |
| case         | 用在 switch 语句中指定控制表达式   | switch (n/10)               |
| catch        | 当异常发生时指定采取的行动          | catch (error)               |
| char         | 一个整数类型                 | char c;                     |
| class        | 指定一个类的声明               | class X { . . . };          |
| compl        | 位运算 NOT 运算符 ~ 的同义词     | b0 = compl b1;              |
| const        | 指定一个常量定义               | const int s = 32;           |
| const_cast   | 用来从不可变的成员函数的内部改变对象     | pp = const_cast<T*>(p)      |
| continue     | 在一个循环中跳到下一次循环的开始       | continue;                   |
| default      | 在一个 switch 语句中的“否则”情况  | default: sum = 0;           |
| delete       | 释放由一个 new 语句分配的内存      | delete a;                   |
| do           | 指定一个 do . . . while 循环 | do { . . . } while . . .    |
| double       | 一个实数类型                 | double x;                   |
| dynamic_cast | 对一个给定的指针, 返回一个 T* 指针   | pp = dynamic_cast<T*>p      |
| else         | 在一个 if 语句中指定一个替代       | else n = 0;                 |
| enum         | 用来声明一个枚举类型             | enum bool { . . . };        |
| explicit     | 用来阻止一个构造函数被含蓄地调用       | explicit X (int n);         |
| export       | 允许从另一个汇编单元访问           | export template<class T>    |
| extern       | 对声明在局部模块外的对象的存储类型      | extern int max;             |
| false        | 布尔类型的其中一种文字            | bool flag = false;          |
| float        | 一种实数类型                 | float x;                    |
| for          | 指定一个 for 循环            | for ( ; ; ) . . .           |
| friend       | 在一个类中指定一个 friend 函数    | friend int f ();            |
| goto         | 引起执行跳到一个标定的语句          | goto error;                 |
| if           | 指定一个 if 语句             | if (n > 0) . . .            |
| inline       | 声明一个函数, 它的文本要被它的调用替代   | inline int f ();            |
| int          | 一个整数类型                 | int n;                      |
| long         | 用来定义整数和实数类型            | long double x;              |
| mutable      | 允许不变的函数来改变域            | mutable string ssn;         |
| namespace    | 允许识别范围模块               | namespace Best { int num; } |

(续表)

| 关键字              | 描述                     | 例子                           |
|------------------|------------------------|------------------------------|
| new              | 分配内存                   | int * p = new int;           |
| not              | NOR 运算符! 的一个同义词        | (not (x==0))                 |
| not_eq           | 不等于运算符!= 的一个同义词        | (x not_eq 0)                 |
| operator         | 用来声明一个重载的运算符           | X operator ++ ()             |
| or               | OR 运算符   的一个同义词        | (x > 0 or x < 8)             |
| or_eq            | 位 OR 赋值运算符 = 的一个同义词    | b1 or_eq b2;                 |
| private          | 在一个类中指定一个 private 声明   | private: int n;              |
| protected        | 在一个类中指定一个 protected 声明 | protected: int n;            |
| public           | 在一个类中指定一个 public 声明    | public: int n;               |
| register         | 在寄存器中存储的对象的存储类型标识符     | register int i;              |
| reinterpret_cast | 返回对象给定的值和类型            | pp = reinterpret_cast<T*>(p) |
| return           | 结束一个函数和返回一个值的语句        | return 0;                    |
| short            | 一个整数类型                 | short n;                     |
| signed           | 用来定义整数类型               | signed char c;               |
| sizeof           | 返回用来存储一个对象的字节数的运算符     | n = sizeof(float);           |
| static           | 在程序期间存在的对象的存储类型        | static int n;                |
| static_cast      | 对一个给定的指针返回一个 T* 指针     | pp = static_cast<T*>p        |
| struct           | 指定一个结构体定义              | struct X { . . . };          |
| switch           | 指定一个 switch 语句         | switch (n) { . . . }         |
| template         | 指定一个 template 类        | template < class T >         |
| this             | 指向当前对象的指针              | return * this;               |
| throw            | 用来产生一个意外               | throw X();                   |
| true             | 布尔类型的其中一种文字            | bool flag = true;            |
| try              | 指定一个包含意外处理的块           | try { . . . }                |
| typedef          | 对一个已存在的类型声明一个同义词       | typedef int Num;             |
| typeid           | 返回一个表示一个表达式的类型的对象      | cout << typeid(x).name();    |
| typename         | 关键字 class 的同义词         | typename X { . . . };        |
| using            | 允许 namespace 前缀冗长的标识   | using namespace std;         |
| union            | 指定一个其元素拥有同样存储的结构       | union z { . . . };           |
| unsigned         | 用来定义整数类型               | unsigned int b;              |
| virtual          | 声明一个在子类中定义的成员函数        | virtual int f();             |
| void             | 指示一个类型的存在              | void f();                    |
| volatile         | 声明一个可以在程序范围外更改的对象      | int volatile n;              |
| wchar_t          | 宽 (16 位) 字符类型          | wchar_t province;            |
| while            | 指定一个 while 循环          | while (n > 0) . . .          |
| xor              | 位排斥 OR 运算符的同义词         | b0 = b1 xor b2;              |
| xor_eq           | 位排斥 OR 赋值运算符= 的同义词     | b1 xor_eq b2;                |

## 附录 C 标准 C++ 运算符

这个表列出了 C++ 中的所有运算符，根据优先权的顺序把它们分组。有更高级别优先权的运算符比较低级别的运算符先估算。例如，在表达式  $(a - b * c)$  中， $*$  运算符将首先估算， $-$  运算符是第二个，因为  $*$  的优先级别是 13，它高于  $-$  的优先级别 12。标志是 “Assoc.” 告诉了一个运算符是右结合的或者是左结合的。例如，表达式  $(a - b - c)$  被估算为  $((a - b) - c)$ ，因为  $-$  是左结合的。标志是 “Arity” 告诉了一个运算符作用于一个、两个或三个操作数（一元的、二元的、或者三元的）。标有 “Overldbl.” 的列告诉了一个运算符是否是可重载的（参见第 8 章）。

| 运算符    | 名字       | 优先级 | 结合性 | 多元性 | 可重载性 | 例子                     |
|--------|----------|-----|-----|-----|------|------------------------|
| ::     | 全局生存空间解析 | 17  | 右   | 一元的 | 否    | <code>t::x</code>      |
| ::     | 类生存空间解析  | 17  | 左   | 二元的 | 否    | <code>X::x</code>      |
| .      | 直接的成员选择  | 16  | 左   | 二元的 | 否    | <code>s.len</code>     |
| ->     | 间接的成员选择  | 16  | 左   | 二元的 | 是    | <code>p-&gt;len</code> |
| []     | 下标       | 16  | 左   | 二元的 | 是    | <code>a[i]</code>      |
| ()     | 函数调用     | 16  | 左   | n/a | 是    | <code>rand()</code>    |
| ()     | 类型构造     | 16  | 左   | n/a | 是    | <code>int(ch)</code>   |
| ++     | 后加       | 16  | 右   | 一元的 | 是    | <code>n++</code>       |
| --     | 后减       | 16  | 右   | 一元的 | 是    | <code>n--</code>       |
| sizeof | 对象或类型的大小 | 15  | 右   | 一元的 | 是    | <code>sizeof(a)</code> |
| ++     | 前加       | 15  | 右   | 一元的 | 是    | <code>++n</code>       |
| --     | 前减       | 15  | 右   | 一元的 | 是    | <code>--n</code>       |
| ~      | 位补码      | 15  | 右   | 一元的 | 是    | <code>s</code>         |
| !      | 逻辑非      | 15  | 右   | 一元的 | 是    | <code>!p</code>        |
| +      | 一元加      | 15  | 右   | 一元的 | 是    | <code>+n</code>        |
| -      | 一元减      | 15  | 右   | 一元的 | 是    | <code>-n</code>        |
| *      | 解除参照     | 15  | 右   | 一元的 | 是    | <code>*p</code>        |
| &      | 地址       | 15  | 右   | 一元的 | 是    | <code>&amp;x</code>    |
| new    | 分配       | 15  | 右   | 一元的 | 是    | <code>new p</code>     |
| delete | 释放       | 15  | 右   | 一元的 | 是    | <code>delete p</code>  |
| ()     | 类型转换     | 15  | 右   | 二元的 | 是    | <code>int(ch)</code>   |
| .      | 直接的成员选择  | 14  | 左   | 二元的 | 否    | <code>x.*q</code>      |
| ->*    | 间接的成员选择  | 14  | 左   | 二元的 | 是    | <code>p-&gt;*q</code>  |
| *      | 乘法       | 13  | 左   | 二元的 | 是    | <code>m*n</code>       |
| /      | 除法       | 13  | 左   | 二元的 | 是    | <code>m/n</code>       |
| %      | 余数       | 13  | 左   | 二元的 | 是    | <code>m%n</code>       |
| +      | 一元加      | 15  | 右   | 一元的 | 是    | <code>+n</code>        |
| -      | 一元减      | 15  | 右   | 一元的 | 是    | <code>-n</code>        |

(续表)

| 运算符    | 名字     | 优先级 | 结合性 | 多元性 | 可重载性 | 例子        |
|--------|--------|-----|-----|-----|------|-----------|
| *      | 解除参照   | 15  | 右   | 一元的 | 是    | *p        |
| &      | 地址     | 15  | 右   | 一元的 | 是    | &x        |
| new    | 分配     | 15  | 右   | 一元的 | 是    | new p     |
| delete | 释放     | 15  | 右   | 一元的 | 是    | delete p  |
| ()     | 类型转换   | 15  | 右   | 一元的 | 是    | int(ch)   |
| .*     | 直接成员选择 | 14  | 左   | 二元的 | 否    | x.*q      |
| ->*    | 间接成员选择 | 14  | 左   | 二元的 | 是    | p->q      |
| *      | 乘法     | 13  | 左   | 二元的 | 是    | m*n       |
| /      | 除法     | 13  | 左   | 二元的 | 是    | m/n       |
| %      | 余数     | 13  | 左   | 二元的 | 是    | m%n       |
| +      | 加法     | 12  | 左   | 二元的 | 是    | m+n       |
| -      | 减法     | 12  | 左   | 二元的 | 是    | m-n       |
| <<     | 位左移    | 11  | 左   | 二元的 | 是    | cout << n |
| >>     | 位右移    | 11  | 左   | 二元的 | 是    | cin >> n  |
| <      | 小于     | 10  | 左   | 二元的 | 是    | x < y     |
| <=     | 小于或等于  | 10  | 左   | 二元的 | 是    | x <= y    |
| >      | 大于     | 10  | 左   | 二元的 | 是    | x > y     |
| >=     | 大于或等于  | 10  | 左   | 二元的 | 是    | x >= y    |
| ==     | 等于     | 9   | 左   | 二元的 | 是    | x == y    |
| !=     | 不等于    | 9   | 左   | 二元的 | 是    | x != y    |
| &      | 位与     | 8   | 左   | 二元的 | 是    | s&t       |
| ^      | 位异或    | 7   | 左   | 二元的 | 是    | s^t       |
|        | 位或     | 6   | 左   | 二元的 | 是    | s t       |
| &&     | 逻辑与    | 5   | 左   | 二元的 | 是    | u&&v      |
|        | 逻辑或    | 4   | 左   | 二元的 | 是    | u  v      |
| ?:     | 条件表达式  | 3   | 左   | 三元的 | 是    | u? x: y   |
| =      | 赋值     | 2   | 右   | 二元的 | 是    | n = 22    |
| +=     | 加法赋值   | 2   | 右   | 二元的 | 是    | n += 8    |
| -=     | 减法赋值   | 2   | 右   | 二元的 | 是    | n -= 4    |
| *=     | 乘法赋值   | 2   | 右   | 二元的 | 是    | n /= 10   |
| /=     | 除法赋值   | 2   | 右   | 二元的 | 是    | n %= 10   |
| %=     | 余数赋值   | 2   | 右   | 二元的 | 是    | s %= mask |
| &=     | 位与赋值   | 2   | 右   | 二元的 | 是    | s &= mask |
| ^=     | 位异或赋值  | 2   | 右   | 二元的 | 是    | s ^= mask |
| =      | 位或赋值   | 2   | 右   | 二元的 | 是    | s  = mask |
| <<=    | 位左移赋值  | 2   | 右   | 二元的 | 是    | s <<= 1   |
| >>=    | 位右移赋值  | 2   | 右   | 二元的 | 是    | s >>= 1   |
| ,      | 逗号     | 0   | 左   | 二元的 | 是    | ++m, --n  |

## 附录 D 标准 C++ 容器类

这个附录总结了标准 C++ 容器类模板和它们最广泛使用的成员函数。这也是标准 C++ 的一部分，它过去叫做标准模板库 (STL)。

### D.1 vector 类模板

一个 vector 对象就像一个有索引范围测试 (使用它的 `at()` 函数) 的数组。作为一个对象，它比数组拥有额外的优势，数组可以被赋值，作为值传递和作为值返回。Vector 类模板在头文件 `<vector>` 中定义。(见例 D.1)

```
vector ();  
// 默认的构造函数：创建一个空的向量  
  
vector (const vector&v);  
// 复制构造函数：创建一个向量 v 的复制  
// 前提条件：*this == v;  
  
vector (unsigned n, const T&x=T ());  
// 构造函数：创建一个包含 n 个元素 x 复制的向量  
// 前提条件：n >= 0;  
// 后续条件：size () == n;  
  
~vector ();  
// 析构函数：摧毁这个向量  
  
vector&operator = (const vector&v);  
// 赋值运算符：把 v 赋给这个向量，使得它是一个复制品  
// 后续条件：*this == v;  
  
unsigned size () const;  
// 返回在这个向量中的元素数目  
  
unsigned capacity () const;  
// 返回这个向量可以容纳而不必再分配的最大元素数目  
  
void reserve (unsigned n);  
// 再分配这个向量，使它的容量为 n 个元素  
// 前提条件：capacity () <= n;  
// 后续条件：capacity () == n;  
  
bool empty () const;
```

```

// 当且仅当 size () == 0 时, 返回 true;

void assign (unsigned n, const T& x = T ());
// 清除这个向量然后插入元素 x 的 n 个复制
// 前提条件: n >= 0;
// 后续条件: size () == n;

T& operator [] (unsigned i);
// 返回元素数目 i;
// 前提条件: 0 <= i < size ();
// 如果前提条件错误的话, 结果是不可预知的

T& at (unsigned i);
// 返回元素数目 i;
// 前提条件: 0 <= i < size ();
// 如果前提条件错误的话, 会有例外产生;

T& front ();
// 返回向量的第一个元素;

T& back ();
// 返回向量的最后一个元素;

iterator begin ();
// 返回一个指向这个向量的第一个元素的迭代符

iterator end ();
// 返回一个指向紧随这个向量最后一个元素的哑元元素的迭代符

reverse_iterator rbegin ();
// 返回一个指向这个向量的最后一个元素的逆迭代符

reverse_iterator rend ();
// 返回一个指现在这个向量的第一个元素前的哑元元素的逆迭代符

void push_back (const T& x);
// 把一个元素 x 的复制加到这个向量的后面
// 前提条件: back () -- x;
// 后续条件: size () has been incremented;

void pop_back (const T& x);
// 删除这个向量的第一个元素
// 前提条件: size () > 0;
// 后续条件: size () 减少了;

iterator insert (iterator p, const T& x);
// 在位置 p 插入元素 x 的一个复制; 返回 p;
// 提前条件: begin () <= p <= end ();
// 后续条件: size () 减少了;

iterator erase (iterator p);
// 删除在位置 p 的元素; 返回 p

```

```
// 前提条件: begin () <= p <= end ();
// 后续条件: size () 减少了;

iterator erase (iterator p1, iterator p2);
// 删除从位置 p1 到位置 p2 前的元素;
// 返回 p1;
// 前提条件: begin () <= p1 <= p2 <= end ();
// 后续条件: size () 已减去 int (p2 - p1);

void clear ();
// 删除向量中的所有元素;
// 前提条件: size () == 0;
```

### 例 D.1 对一个向量对象使用迭代符

```
#include <iostream>

#include <vector>
using namespace std;
typedef vector<int>::iterator It;

int main ()
{ vector<int> v (4);
  for (int i = 0; i < 4; i++)
    v[i] = 222 * i + 333;
  cout << "Using the iterator it in a for loop: \n";
  for (It it = v.begin (); it != v.end (); it++)
    cout << "\t * it = " << *it << "\n";
  cout << "Using the iterator p in a while loop: \n";
  It p = v.begin ();
  while (p != v.end ())
    cout << "\t * p++ = " << *p++ << "\n";
}
```

Using the iterator it in a for loop:

```
* it = 333
* it = 555
* it = 777
* it = 999
```

Using the iterator p in a while loop:

```
* p++ = 333
* p++ = 555
* p++ = 777
* p++ = 999
```

向量 v 有 4 个元素: 333、555、777 和 999。第二个 for 循环应用迭代符来从头到尾遍历向量 v, 用 \* it 访问它的每个元素。这个 while 循环和使用 \* p 的效果一样。

### 例 D.2 对向量对象使用迭代符

```
#include <iostream>

#include <vector>
using namespace std;
```

```

typedef vector<int>::reverse_iterator Rit;

int main ()
{ vector<int> v (4);
  for (int i=0; i<4; i++)
    v[i] = 222*i + 333;
  cout << "Using the reverse iterator rit in a for loop: \n";
  for (Rit rit=v.rbegin(); rit!=v.rend(); rit++)
    cout << "\t*rit=" << *rit << "\n";
  cout << "Using the reverse iterator rp in a while loop: \n";
  Rit rp=v.rbegin();
  while (rp!=v.rend())
    cout << "\t*rp++=" << *rp++ << "\n";
}

```

```

Using the reverse iterator rit in a for loop:
    *rit = 999
    *rit = 777
    *rit = 555
    *rit = 333
Using the reverse iterator rp in a while loop:
    *rp++ = 999
    *rp++ = 777
    *rp++ = 555
    *rp++ = 333

```

向量  $v$  有 4 个元素：333、555、777 和 999（和例 D.1 相同）。第二个 for 循环使用遍历迭代符  $rit$  来向后遍历向量  $v$ ，用  $*rit$  访问它的每个元素。这个 while 循环和使用  $*rp$  的效果一样。

### 例 D.3 对一个向量对象使用 insert () 函数

```

#include <iostream>

#include <vector>
using namespace std;
typedef vector<int> Vector;
typedef Vector::iterator It;
void print (const Vector&);

int main ()
{ Vector v (4);
  for (int i=0; i<4; i++)
    v[i] = 222*i + 333;
  print (v);
  It it = v.insert (v.begin () + 2, 666);
  print (v);
  cout << "*it=" << *it << "\n";
}

void print (const Vector& v)
{ cout << "size=" << v.size () << ": (" << v[0];

```



```

    for (int i = 1; i < v.size(); i++)
        cout << "," << v[i];
    cout << ")" << "\n";
}
size=4: (333, 555, 777, 999)
size=5: (333, 555, 666, 777, 999)
* it = 666

```

向量  $v$  有 4 个元素：333, 555, 777 和 999（和例 D.1 相同）。第二个 for 循环使用遍历迭代符  $rit$  来向后遍历向量  $v$ ，用  $*rit$  访问它的每个元素。这个 while 循环和使用  $*rp$  的效果一样。

#### 例 D.4 对一个向量对象使用一些通用的算法

```

#include <iostream>
#include <vector>
using namespace std;
typedef vector<int> Vector;
typedef Vector::iterator It;
void print (const Vector&);

int main ()
{
    Vector v (9);
    for (int i = 0; i < 9; i++)
        v[i] = 111 * i + 111;
    print (v);
    It it = v.begin();
    fill (it + 2, it + 5, 400); //用 400 代替 v[2: 5]
    print (v);
    reverse (it + 4, it + 7); //
    print (v);
    iter_swap (it + 6, it + 8);
    print (v);
    sort (it + 4, it + 9);
    print (v);
}

void print (const Vector& v)
{
    cout << "size=" << v.size() << ": (" << v[0];
    for (int i = 1; i < v.size(); i++)
        cout << "," << v[i];
    cout << ")" << "\n";
}
size=9: (111, 222, 333, 444, 555, 666, 777, 888, 999)
size=9: (111, 222, 400, 400, 400, 666, 777, 888, 999)
size=9: (111, 222, 400, 400, 777, 666, 400, 888, 999)
size=9: (111, 222, 400, 400, 777, 666, 999, 888, 400)
size=9: (111, 222, 400, 400, 400, 666, 777, 888, 999)

```

#### 例 D.5 对一个向量对象使用一些更加一般的算法

```

#include <iostream>

```

```

#include <vector>
using namespace std;
typedef vector<int> Vector;
typedef Vector::iterator It;
void print (const Vector&);

int main ()
{ Vector v1 (9);
  for (int i=0; i<9; i++)
    v1[i] = 111*i + 111;
  print (v1);
  Vector v2 (9);
  print (v2);
  It p1 = v1.begin (), p2 = v2.begin ();
  copy (p1+3, p1+8, p2+3);
  print (v2);
  It p = min_element (p1+4, p1+8);
  cout << "*p=" << *p << "\n";
  p = max_element (p1+4, p1+8);
  cout << "*p=" << *p << "\n";
  p = find (p1, p1+9, 444);
  if (p != p1+9) cout << "*p=" << *p << "\n";
}

void print (const Vector& v)
{ cout << "size=" << v.size () << ": (" << v[0];
  for (int i=1; i<v.size (); i++)
    cout << "," << v[i];
  cout << ") \n";
}

size=9: (111, 222, 333, 444, 555, 666, 777, 888, 999)
size=9: (0, 0, 0, 0, 0, 0, 0, 0, 0)
size=9: (0, 0, 0, 444, 555, 666, 777, 888, 0)
*p=555
*p=888
*p=444

```

## D.2 deque 类模板

一个 deque (发音“deck”) 对象就是一个双端队列, 是为了在它的开始和结尾提供有效的插入和删除。除了向量类有的所有成员函数外 (除了 capacity () 函数和 reserve () 函数外), 它还有下面的两个成员函数。deque 类模板在 <deque> 头文件中定义。

```

void push_front (const T& x);
// 在队列的头部插入元素 x 的一个复制
// 前提条件: front () == x;
// 后续条件: size () 已减少了

void pop_front ();

```

```
// 删除这个向量的第一个元素
// 前提条件: size() > 0;
// 后续条件: size() 已减少了
```

## D.3 stack 类模板

一个 stack 对象是一个序列容器，它只允许在一个叫做“栈顶”的末端进行插入和删除。在标准 C++ 库中，stack 类模板是从 deque 类模板改进的。这意味着 stack 的成员函数是随着 deque 的成员函数实现的，如下所示。stack 类模板在 <stack> 文件中定义。

```
template < class T > class stack
{
public:
    unsigned size() const { return _d.size(); }
    bool empty() const { return _d.empty(); }
    T& top() { return _d.back(); }
    void push(const T& x) { _d.push_back(x); }
    void pop() { _d.pop_back(); }
protected:
    deque<T> _d;
};
```

## D.4 queue 类模板

一个 queue 对象是一个序列容器，它只允许在一端末端进行插入和另一端末端进行删除。像 stack 类模板，在标准 C++ 库中，queue 类模板是从 deque 类模板改进的。这意味着 queue 的成员函数是随着 deque 的成员函数实现的，如下所示。queue 类模板在 <queue> 文件中定义。

```
template < class T > class queue
{
public:
    unsigned size() const { return _d.size(); }
    bool empty() const { return _d.empty(); }
    T& front() { return _d.front(); }
    T& back() { return _d.back(); }
    void push(const T& x) { _d.push_back(x); }
    void pop() { _d.pop_front(); }
protected:
    deque<T> _d;
};
```

## D.5 priority\_queue 类模板

一个 priority\_queue 对象是一个容器类，它就像一个队列，除了元素弹出的顺序由它的优先权决定外。这意味着必须为类型 T 的元素定义函数 operator<()。这个 priority\_queue

类模板在 `<queue>` 头文件中定义。(见例 D.6)

```
vector ();
// 构造一个空的向量

vector (const vector& v);
// 构建一个向量 v 的拷贝
// 后续条件: *this == v;
```

### 例 D.6 使用一个 `priority_queue` 对象

```
#include <iostream>

#include <queue>
using namespace std;
int main ()
{ priority_queue<string> pq;
  pq.push ("Japan");
  pq.push ("Japan");
  pq.push ("Korea");
  pq.push ("China");
  pq.push ("India");
  pq.push ("Nepal");
  pq.push ("Qatar");
  pq.push ("Yemen");
  pq.push ("Egypt");
  pq.push ("Zaire");
  pq.push ("Libya");
  pq.push ("Italy");
  pq.push ("Spain");
  pq.push ("Chile");
  while (! pq.empty ())
  { cout << pq.top () << " \n";
    pq.pop ();
  }
}
```

```
Zaire
Yemen
Spain
Qatar
Nepal
Libya
Korea
Japan
Japan
Italy
India
India
Egypt
China
Chile
```

这个优先权队列总是把它的最高优先级元素先权保留在队列的头部（也就是前面）。由于使

用了标准的串的词典排序法（也就是字典排序法），结果就是以相反的字母顺序被访问的名字。

注意，`priority_queue` 对象存储完全相同的元素。

## D.6 list 类模板

一个列表对象就是一个序列容器，它允许在序列中的任何位置进行有效的插入和删除。除了 `deque` 类具有的所有成员函数（除了 `operator[]()` 和 `at()` 函数外）外，它还有下面的成员函数。列表类模板在 `<list>` 头文件中定义。

```
void splice ( iterator p, list&l, iterator p1 );
// 从 l 中的位置 p1 移动元素到这个列表的位置 p;
// 前提条件: p 是这个列表中的一个正确的迭代符;
// 后续条件: p1 是列表 l 中的一个正确的迭代符;

void splice ( iterator p, list&l, iterator p1, iterator p2 );
// 从 l 中的位置 [p1: p2-1] 移动元素到以位置 p 开始的列表;
// 确定前提条件: p 是这个列表中的一个正确的迭代符;
// 后续条件: p1 和 p2 是列表 l 中的一个正确的迭代符;
// 后续条件: p1 < p2;

void remove ( const T&x );
// 从这个列表中删除等于 x 的所有元素;
// 不变量: 所有没有删除的元素的顺序;
// 不变量: 所有指向没有删除的元素的迭代符

void unique ( );
// 从这个列表中删除所有复制的元素;
// 不变量: 所有没有删除的元素的顺序;
// 不变量: 所有指向没有删除的元素的迭代符;

void merge ( list&l );
// 把列表 l 中的所有元素合并到这个列表中;
// 前提条件: 列表 l 和这个列表都是已排序的;
// 前提条件: size() 增加 l.size();
// 复杂性: O(n);

void reverse ( );
// 把这个列表的元素的顺序颠倒;
// 不变量: size();
// 复杂性: O(n);

void sort ( );
// 排序这个列表中的元素;
// 后续条件: 这个列表是已排序的;
// 不变量: size();
// 复杂性: O(n*log(n));
```

## 例 D.7 排序和逆序一个列表对象

```

#include <iostream>
#include <list>
using namespace std;
typedef list<string> List;
typedef List::iterator It;
void print (List&);

int main ()
{ List l;
  l.push_back ("Kenya");
  l.push_back ("Sudan");
  l.push_back ("Egypt");
  l.push_back ("Zaire");
  l.push_back ("Libya");
  l.push_back ("Congo");
  l.push_back ("Ghana");
  print (l);
  l.sort ();
  print (l);
  l.reverse ();
  print (l);
}

void print (List& l)
{ cout << "\n";
  for (It it = l.begin (); it != l.end (); it++)
    cout << *it << " ";
}

```

Kenya  
Sudan  
Egypt  
Zaire  
Libya  
Congo  
Ghana

Congo  
Egypt  
Ghana  
Kenya  
Libya  
Sudan  
Zaire  
Zaire  
Sudan  
Libya  
Kenya  
Ghana  
Egypt  
Congo

## D.7 map 类模板

一个映射对象（叫做一个字典，一个表，一个相关联数组）就像一个数组，它的索引可以是实现 `<` 运算符的任何类型。一个映射就像一人数学函数，它对每一个 `x` 值，都给出唯一的 `y` 值。`x` 值，叫做关键字值，就是索引。`y` 值是关键字所标识的存储对象。

一个英语字典就是一个映射对象的例子。关键字就是单词，它的相关联对象是这个单词的字典定义。

另一个标准的例子就是一个学生记录的数据库表。关键字就是学生的身份证号（也就是社会保障号），它的相关联对象是这个学生的数据记录。

`map` 类模板在 `<map>` 头文件中定义。它和 `vector` 类模板具有相同的成员函数。

### 例 D.8 使用一个映射对象

```
#include <iostream>

#include <map>
using namespace std;

struct Country
{ friend ostream& operator<< (ostream&, const Country&);
  Country ();
  Country (string, string, string, int, int);
  string abbr, capital, language;
  int population, area;
};

typedef map< string, Country> Map;
typedef Map:: iterator It;
typedef pair< const string, Country> Pair;
void load (Map&);
void print (Map&);
void find (Map&, const string&);

int main ()
{ Map map;
  load (map);
  print (map);
  find (map, "Cuba");
  find (map, "Iran");
  find (map, "Oman");
}

ostream& operator<< (ostream& ostr, const Country& c)
{ return ostr << c.abbr << ", " << c.capital << ", " << c.language
  << ", pop = " << c.population << ", area = " << c.area;
}

Country:: Country ()
```

```

: abbr (""), capital (""), language (""), population (0), area (0)

Country::Country (string an, string c, string l, int p, int a)
: abbr (ak), capital (c), language (l), population (p), area (a) {}

void load (Map& m)
{ m["Iran"] = Country ("IR", "Tehran", "Farsi", 68959931, 632457);
  m["Iran"] = Country ("IR", "Tehran", "Farsi", 68959931, 632457);
  m["Peru"] = Country ("PE", "Lima", "Spanish", 26111110, 496223);
  m["Iraq"] = Country ("IQ", "Baghdad", "Arabic", 21722278, 167975);
  m.insert (Pair ("Togo", Country ("TG", "Lome", "French", 4905824, 21927)));
  m.insert (Pair ("Fiji", Country ("FJ", "Suva", "English", 802611, 7054)));
  m.insert (Pair ("Fiji", Country ("FJ", "Suva", "Fijian", 802611, 7054)));
}

void print (Map& m)
{ for (it = m.begin(); it != m.end(); it++)
  cout << it->first << ": " << it->second << " " << endl;
  cout << "size=" << m.size() << " " << endl;
}

void find (Map& m, const string& s)
{ cout << s << endl;
  it = m.find (s);
  if (it == m.end()) cout << "was not found. " << endl;
  else cout << ": " << it->second << " " << endl;
}

Fiji:    FJ, Suva, English, pop= 802611, area= 7054
Iran:    IR, Tehran, Farsi, pop= 68959931, area= 632457
Iraq:    IQ, Baghdad, Arabic, pop= 21722278, area= 167975
Peru:    PE, Lima, Spanish, pop= 26111110, area= 496223
Togo:    TG, Lome, French, pop= 4905824, area= 21927
Size= 5
Cuba was not found.
Iran:    IR, Tehran, Farsi, pop= 68959931, area= 632457
Qwen was not found.

```

程序创建了一个映射，它的关键字是四个字母的国家名字，它的映射值是 Country 对象，此处的 Country 是一个有五个字段的类：abbr（缩写）、capital（首都）、language（语言）、population（人口）和 area（面积）。它使用一个单独的函数把数据装载入映射中。

load（）函数阐明了两种不同的方式把一对元素插入到一个映射中。前四行使用下标运算符，后三行使用了 insert（）函数。对于一个映射容器，下标运算符对一个映射容器的作用和其他容器类的作用相同：就像一个数组，除了一个映射的索引不必是一个整数外。在这个例子中它是一个串。

insert（）函数只带一对参数，这两个成员的类型对映射本身必须是相同的，除了第一个成员（关键字字段）必需是常量外。

映射类不允许相同的关键字。注意当一个相同的关键字插入时，下标运算符替代了存在的元素，使得最后一对插入的元素就是保留的那对。但是当同一个相同的关键字插入时



insert () 函数并不替代存在的元素, 因此第一对插入的元素就是保留的那对。

print () 函数使用迭代符 it 来遍历映射。对 for 循环中的每次循环, 它指向一对对象, 其第一个成员是关键字值, 第二个成员是数据对象。这两个成员被表达式 it->first 和 it->second 访问。第一个成员是一个四个字母的国家名字的串, 第二个成员是一个 Country 对象, 由于它在 Country 类定义中被重载, 它能被传递给输出运算符。注意这对成员根据它们的关键字值排序。

find () 函数使用了映射类中的 find 成员函数。调用 m.find (s) 返回一个迭代符, 它指向第一个成员等于 s 的映射元素。如果没找到这样的元素, 那么返回的指针指向 m.end (), 它是紧随映射容器的最后一个元素的哑元元素。

## D.8 set 类模板

一个集合对象就像一个只有存储的关键字的映射对象。

set 类模板在头文件 <set> 中定义。

### 例 D.9 使用集合函数

这个程序定义了重载的 +、\* 和 - 来演示集合理论的合并, 交集和相关的成员运算。这些是通过使用 insert () 和 erase () 成员函数、set\_intersection () 和 set\_difference () 集合的一般算法 (set\_union、set\_difference () 和 set\_difference ()) 和相应的集合理论的运算 (合并、交集和补集) 实现的。

```
#include <iostream>
#include <set>
#include <string>
using namespace std;
typedef set<string> Set;
typedef set<string>::iterator It;
void print (Set);
Set operator + (Set&, Set&); // 合并
Set operator * (Set&, Set&); // 交集
Set operator - (Set&, Set&); // 补集

int main ()
{
    string str1 [] = { "A", "B", "C", "D", "E", "F", "G" };
    string str2 [] = { "A", "E", "I", "O", "U" };
    Set s1 (str1, str1 + 7);
    Set s2 (str2, str2 + 5);
    print (s1);
    print (s2);
    print (s1 + s2);
    print (s1 * s2);
    print (s1 - s2);
}
```

```

Set operator + (Set& s1, Set& s2)
{ Set s (s1);
  s.insert (s2.begin (), s2.end ());
  return s;
}

Set operator * (Set& s1, Set& s2)
{ Set s (s1);
  for (it = set_intersection (s1.begin (), s1.end (),
                              s2.begin (), s2.end (), s.begin ());
       s.erase (it, s.end ());
       return s;
}

Set operator - (Set& s1, Set& s2)
{ Set s (s1);
  for (it = set_difference (s1.begin (), s1.end (),
                           s2.begin (), s2.end (), s.begin ());
       s.erase (it, s.end ());
       return s;
}

void print (Set s)
{ cout << "size=" << s.size () << ": ";
  for (it = s.begin (); it != s.end (); it++)
    if (it == s.begin ()) cout << *it;
    else cout << ", " << *it;
  cout << "\n";
}

size=7:    {A, B, C, D, E, F, G}
size=5:    {A, E, I, O, U}
size=10:   {A, B, C, D, E, F, G, I, O, U}
size=2:    {A, E}
size=5:    {B, C, D, F, G}

```

集合对象  $s_1$  和  $s_2$  是使用表达式  $str_1$ 、 $str_1 + 7$ 、 $str_2$  和  $str_2 + 7$  作为迭代符，从串队列  $str_1$  和  $str_2$  构造的。

一个集合的元素总是按分类的顺序存储的。这允许合并函数 ( $\text{operator} + ()$ ) 用  $\text{set::insert}()$  函数来实现。

集合一般的算法为什么不能直接生成预期的集合理论的运算的主要原因是，集合一般算法剩下的目标集合的大小没有变化。因此，使用  $\text{erase}()$  函数和  $\text{set\_intersection}()$ ， $\text{set\_difference}()$  一般算法来实现  $\text{operator} *()$  和  $\text{operator} -()$  函数。

## 附录 E 标准 C++ 一般算法

标准 C++ 中的一般算法是应用于容器对象的 70 个非成员函数模板。这里是按字母顺序列出的 66 个。我们使用符号  $[p, q[$  来表示从  $*p$  到  $*(q-1)$  的元素段（也就是说，包括元素  $p$  但不包括元素  $*q$ ）。参数是：

```
iterator p, q;      // 用来描述段  $[p, q[$ 
iterator r;         //  $p \leq r \leq q$ 
unsigned n;         // 用做计数器
T& x, y;            // 序列的元素类型值
class p;            // 一个谓词类，带有布尔型的 operator ()()
```

参数列表  $(p, q, pp)$  用的最频繁；它的意思是从段  $[p, q[$  中的元素要被复制到段  $[pp, pp+n[$ ，此处的  $n$  是在  $[p, q[$  中的元素数，也就是  $q-p$ 。

为简单，我们使用数组代替一般的容器对象。在这种情况下，指针代替了指示符。回想一下，如果  $a$  是一个数组， $k$  是一个  $\text{int}$  型，则  $a+k$  代表了从  $a[k]$  开始的子数组，并且  $*(a+k) = a[k]$ 。如果  $l$  是数组的长度，则  $a+l$  指向数组最后一个成员后的（虚）成员。

下面的  $\text{Print}()$  函数用于显示数组  $a$  中的  $n$  个成员  $a[0], \dots, a[n-1]$ ：

```
void print (int * a, int n)
{ cout << "n=" << n << "; " << a[0];
  for (int i=1; i<n; i++)
    cout << ", " << a[i];
  cout << "\n";
}
```

这里列出的 66 个算法自然得分为 8 组，在下表中总结：

### < algorithm > 中的搜索和排序算法

|                                |                           |
|--------------------------------|---------------------------|
| <code>binary_search()</code>   | 判断一个所给的值是否是段中的一个成员        |
| <code>inplace_merge()</code>   | 将两个相近的已排序段合并成一个已排序段       |
| <code>lower_bound()</code>     | 寻找段中第一个与给定值相同的成员          |
| <code>merge()</code>           | 将两个已排序段合并到第三个已排序段中        |
| <code>nth_element()</code>     | 寻找给定值第一次出现的地方             |
| <code>partial_sort()</code>    | 将段中的前 $n$ 个成员排序           |
| <code>partial_sort_copy</code> | 复制段中最小的 $n$ 个成员到另一个已排序段中  |
| <code>partition()</code>       | 分割段，使第一部分中的成员满足 $P(x)$ 为真 |
| <code>sort()</code>            | 将段排序                      |
| <code>upper_bound()</code>     | 寻找段中最后一个与给定值相同的成员         |

**< algorithm > 中关于序列的非改进算法**

|                              |                    |
|------------------------------|--------------------|
| <code>adjacent_find()</code> | 寻找段中第一个相领的对        |
| <code>count()</code>         | 记数与给定值相同的成员个数      |
| <code>count_if()</code>      | 记数满足给定条件的成员个数      |
| <code>equal()</code>         | 判断两个成员是否有相同的值且顺序相同 |
| <code>find()</code>          | 寻找第一个与给定值相同的成员     |
| <code>find_end()</code>      | 寻找最后一个与给定值相同的成员    |
| <code>find_first_of()</code> | 寻找给定字符串第一次出现的位置    |
| <code>find_if()</code>       | 寻找第一个满足给定条件的成员     |
| <code>for_each()</code>      | 对每一个成员执行一个函数       |
| <code>mismatch()</code>      | 寻找两个段第一个不相同的位置     |
| <code>search()</code>        | 寻找一个后继串            |
| <code>search_n()</code>      | 寻找 n 个具有相同值的后继串    |

**< algorithm > 中关于序列的改进算法**

|                                |                                   |
|--------------------------------|-----------------------------------|
| <code>copy()</code>            | 将段复制到另一个位置                        |
| <code>copy_backward()</code>   | 将段复制到另一位置                         |
| <code>fill()</code>            | 将段中所有成员赋给定值                       |
| <code>fill_n()</code>          | 将段中 n 个成员赋给定值                     |
| <code>generate()</code>        | 将段中所有成员用 f(x) 函数进行处理, 并传递给输出      |
| <code>generate_n()</code>      | 将段中所有成员用 f(x) 函数进行 n 次处理, 并传递给输出  |
| <code>iter_swap()</code>       | 交换给定指示符所指位置处的成员                   |
| <code>random_shuffle()</code>  | 将段中成员打乱顺序                         |
| <code>remove()</code>          | 将所有与给定值不同的成员向左移                   |
| <code>remove_copy()</code>     | 将所有与给定值不同的成员复制到另一个段中              |
| <code>remove_copy_if()</code>  | 将所有满足 P(x) 为假的成员复制到另一个段中          |
| <code>remove_if()</code>       | 将所有满足 P(x) 为假的成员向左移               |
| <code>replace()</code>         | 将段中所有值为 x 的成员改为 y                 |
| <code>replace_copy()</code>    | 将段中所有值为 x 的成员复制到另一个段中, 并改为 y      |
| <code>replace_copy_if()</code> | 将段中所有满足 P(x) 为真的成员复制到另一个段中, 并改为 y |
| <code>replace_if()</code>      | 将段中所有满足 P(x) 为真的成员值改为 y           |
| <code>reverse()</code>         | 将段中成员反转                           |
| <code>reverse_copy()</code>    | 将段中成员以反转顺序复制到另一个段中                |
| <code>swap()</code>            | 交换两个成员                            |
| <code>transform()</code>       | 对所有成员用 f(x) 进行处理, 并重新排序           |
| <code>unique()</code>          | 将每个出现的值左移                         |
| <code>unique_copy()</code>     | 复制所有不相同的成员到另一个段中                  |

**< algorithm > 中的比较算法**

|                                        |                          |
|----------------------------------------|--------------------------|
| <code>lexicographical_compare()</code> | 如果第一个成员按字典顺序比第二个成员小, 返回真 |
| <code>max()</code>                     | 返回段中最大成员的值               |
| <code>max_element()</code>             | 返回段中最大成员的位置              |
| <code>min()</code>                     | 返回段中最小成员的值               |
| <code>min_element()</code>             | 返回段中最小成员的位置              |

**< algorithm > 中关于集合的算法**

|                                          |                         |
|------------------------------------------|-------------------------|
| <code>includes ()</code>                 | 如果第一个段中包含第二个段中所有成员, 返回真 |
| <code>set_difference ()</code>           | 将两个段的不同之处复制到第三个中        |
| <code>set_intersection ()</code>         | 将两个段的交集复制到第三个中          |
| <code>set_symmetric_difference ()</code> | 将两个段不对称处复制到第三个中         |
| <code>set_union ()</code>                | 将两个段的并集复制到第三个中          |

**< algorithm > 中关于堆的算法**

|                           |                                                     |
|---------------------------|-----------------------------------------------------|
| <code>make_heap ()</code> | 将段中成员重新在堆中排序                                        |
| <code>pop_heap ()</code>  | 将堆中第一个成员移到最后, 并重新用 <code>make_heap</code> 排序        |
| <code>push_heap ()</code> | 将最后一个成员左移, 使段成为堆                                    |
| <code>sort_heap ()</code> | 使用 <code>n</code> 次 <code>pop_heap ()</code> , 对堆排序 |

**< algorithm > 中的排列算法**

|                                  |                                                       |
|----------------------------------|-------------------------------------------------------|
| <code>next_permutation ()</code> | 改变段顺序, <code>n!</code> 次调用可以得到 <code>n!</code> 次不同的顺序 |
| <code>prev_permutation ()</code> | 改变段顺序, <code>n!</code> 次调用可以得到 <code>n!</code> 次不同的顺序 |

**< numeric > 中的数字算法**

|                                     |                                    |
|-------------------------------------|------------------------------------|
| <code>accumulate ()</code>          | 将段中所有成员相加, 返回 <code>x + sum</code> |
| <code>adjacent_difference ()</code> | 将第一个段中相邻成员的和放入第二段中                 |
| <code>inner_product ()</code>       | 返回两个段的内积                           |
| <code>partial_sum ()</code>         | 将第一个段的局部和放入第二个段中                   |

寻找一个成员的算法总是返回指向这个成员的指示符, 或者指向这个序列最后一个成员后面的虚成员。

使用判断的算法用下面的判断类说明:

```
class Odd
{
public:
    bool operator () (int n) { return n%2 ? true : false; }
};
```

这个类像函数一样被传递, 如: `Odd ()`。(见例 E.8)

注意修改算法不改变段的长度 `[p, q]`。它们返回一个指示符指向被修改成员的后面部分。

```
accumulate (p, q, x):
// 返回 x 加上段 [p, q] 中元素和的值;
// 不变量: [p, q] 未变化;
```

**例 E.1 测试 `accumulate ()` 算法**

```
int main ()
{
    int a [] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
    int sum = accumulate (a, a + 10, 1000);
    cout << "sum = " << sum << '\n';
}

sum = 1088
```

```
adjacent_difference(p, q, pp);
//用 b[i] = a[i] - a[i-1] 装载段 a[pp, pp+p-q];
//不变量: [p, q] 不变化;
```

### 例 E.2 测试 adjacent\_difference() 算法

```
int main()
{
    int a[] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34};
    print(a, 10);
    int b[10];
    adjacent_difference(a, a+10, b);
    print(b, 10);
}
n=10: {0, 1, 1, 2, 3, 5, 8, 13, 21, 34}
n=10: {0, 1, 0, 1, 1, 2, 3, 5, 8, 13}
```

adjacent\_difference() 算法是 partial\_sum() 的逆算法 (见例 E.36)

```
adjacent_find(p, q);
//返回段 a[p, q] 中和它的后继元素有相同值的第一个元素的位置;
//不变量: [p, q] 不变化;
```

### 例 E.3 测试 adjacent\_find() 算法

```
int main()
{
    int a[] = {0, 1, 0, 1, 1, 1, 0, 1, 1, 0};
    print(a, 10);
    int *r = adjacent_find(a, a+10);
    cout << " *r=" << *r << '\n'; // 这是元素 a[i]
    cout << "r-a=" << r-a << '\n'; // 这是下标;
}
n=10: {0, 1, 0, 1, 1, 1, 0, 1, 1, 0}
*r=1
r-a=3
```

```
binary_search(p, q, x);
//当且仅当 x 在段 [p, q] 中时, 返回 true;
//前提条件: 段 [p, q] 必须是已排序的;
//不变量: [p, q] 不变化;
```

### 例 E.4 测试 binary\_search() 算法

```
int main()
{
    int a[] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34};
    print(a, 10);
    bool found = binary_search(a, a+10, 21);
    cout << "found=" << found << '\n';
    found = binary_search(a+2, a+7, 21);
```

```

    cout << "found=" << found << '\n';
}
n=10: {0, 1, 1, 2, 3, 5, 8, 13, 21, 34}
found=1
found=0

```

```

copy (p, q, pp);
//复制段 [p, q [ 到 [pp, pp+n [, 此时 n=q-p;
//不变量: [p, q [ 不变化;

```

### 例 E.5 测试 **copy** () 算法

```

int main ()
{ int a [] = {100, 111, 122, 133, 144, 155, 166, 177, 188, 199};
  print (a, 10);
  copy (a+7, a+10, a+2);
  print (a, 10);
  int b [3];
  copy (a+7, a+10, b);
  print (b, 3);
}
n=10: {100, 111, 122, 133, 144, 155, 166, 177, 188, 199}
n=10: {100, 111, 177, 188, 199, 155, 166, 177, 188, 199}
n=3: {177, 188, 199}

```

```

copy_backward (p, q, pp);
//复制段 [p, q [ 到 [pp-n, pp[, 此时 n=q-p;
//不变量: [p, q [ 不变化;

```

### 例 E.6 测试 **copy\_backward** () 算法

```

int main ()
{ int a [] = {100, 111, 122, 133, 144, 155, 166, 177, 188, 199};
  print (a, 10);
  copy_backward (a+7, a+10, a+5);
  print (a, 10);
  int b [3];
  copy_backward (a+7, a+10, b+3);
  print (b, 3);
}
n=10: {100, 111, 122, 133, 144, 155, 166, 177, 188, 199}
n=10: {100, 111, 177, 188, 199, 155, 166, 177, 188, 199}
n=3: {177, 188, 199}

```

```

count (p, q, x);
//返回段 [p, q [ 中 x 出现的次数;
//不变量: [p, q [ 不变化;

```

### 例 E.7 测试 **count** () 算法

```

int main ()

```

```

int a[] = {0, 1, 0, 1, 1, 1, 0, 1, 1, 0};
print (a, 10);
int r = count (a, a+10, 1);
cout << "n=" << n << '\n';
}
n=10: {0, 1, 0, 1, 1, 1, 0, 1, 1, 0}
n=6

```

**count\_if (p, q, p1):**  
 //返回段 [p, q] 中 p1(x) 出现的次数;  
 //不变量: [p, q] 不变化;

### 例 E.8 测试 count\_if () 算法

```

int main ()
{
  int a[] = {0, 1, 0, 1, 1, 1, 0, 1, 1, 0};
  print (a, 10);
  int n = count_if (a, a+10, Odd ());
  cout << "n=" << n << '\n';
}

n=10: {0, 1, 0, 1, 1, 1, 0, 1, 1, 0}
n=6

```

**equal (p, q, pp):**  
 //当且仅当段 [p, q] 和段 [pp, pp+n] 匹配时返回 true, 此时 n = q - p;  
 //不变量: [p, q] 和 [pp, pp+n] 不变化;

### 例 E.9 测试 equal () 算法

```

int main ()
{
  int a[] = {0, 1, 0, 1, 1, 1, 0, 1, 1, 0};
  int b[] = {0, 1, 0, 0, 1, 1, 0, 1, 0, 0};
  print (a, 10);
  print (b, 10);
  cout << "equal (a, a+10, b) =" << equal (a, a+10, b) << '\n';
  cout << "equal (a+1, a+4, a+5) =" << equal (a+1, a+4, a+5) << '\n';
}

n=10: {0, 1, 0, 1, 1, 1, 0, 1, 1, 0}
n=10: {0, 1, 0, 0, 1, 1, 0, 1, 0, 0}
equal (a, a+10, b) = 0
equal (a+1, a+4, a+5) = 1

```

**fill (p, q, x):**  
 //用 x 代替段 [p, q] 中的每个元素;

### 例 E.10 测试 fill () 函数

```

int main ()

```



```

{ int a [] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34};
  print (a, 10);
  fill (a+6, a+9, 0);
  print (a, 10);
}

```

```

n=10: {0, 1, 1, 2, 3, 5, 8, 13, 21, 34}
n=10: {0, 1, 1, 2, 3, 5, 0, 0, 0, 34}

```

**fill** *n* (*p*, *n*, *x*);  
 //用*x*代替段 [*p*, *p*+*n*] 中的每个元素;

### 例 E.11 测试 fill\_n () 算法;

```

int main ()
{ int a [] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34};
  print (a, 10);
  fill_n (a+6, 3, 0);
  print (a, 10);
}

```

```

n=10: {0, 1, 1, 2, 3, 5, 8, 13, 21, 34}
n=10: {0, 1, 1, 2, 3, 5, 0, 0, 0, 34}

```

**find** (*p*, *q*, *x*);  
 //返回段 [*p*, *p*+*n*] 中 *x* 的第一个位置;  
 //不变量: [*p*, *q*] 不变化;

### 例 E.12 测试 find () 算法

```

int main ()
{ int a [] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34};
  print (a, 10);
  int * r = find (a, a+10, 13);
  cout << " *r=" << *r << '\n'; // 这是元素 a [i]
  cout << "r-a=" << r-a << '\n'; // 这是下标 i
  r = find (a, a+6, 13);
  cout << " *r=" << *r << '\n'; // 这是元素 a [i]
  cout << "r-a=" << r-a << '\n'; // 这是下标 i
}

```

```

n=10: {0, 1, 1, 2, 3, 5, 8, 13, 21, 34}
*r=13
r-a=7
*r=8
r-a=6

```

**find\_end** (*p*, *q*, *pp*, *qq*);  
 //返回在段 [*p*, *q*] 中段 [*pp*, *qq*] 最后一次出现的位置;

//不变量:  $[p, q]$  和  $[pp, qq]$  不变化;

### 例 E.13 测试 find\_end () 算法

```
int main ()
{
    int a [] = {0, 1, 0, 1, 1, 1, 0, 1, 1, 0};
    int b [] = {1, 0, 1, 1, 1};
    int * r = find_end (a, a+10, b, b+5);    // 在 a 中查找 10111
    cout << " *r=" << r << '\n';           // 这是元素 a [1]
    cout << "r-a=" << r-a << '\n';          // 这是下标 i
    r = find_end (a, a+10, b, b+4);           // 在 a 中查找 1011
    cout << " *r=" << *r << '\n';
    cout << "r-a=" << r-a << '\n';
}

*r=1
r-a=1
*r=1
r-a=5
```

find\_first\_of (p, q, pp, qq);

//返回在  $[p, q]$  中的第一个元素的位置, 并且它也在  $[pp, qq]$  中;

//不变量:  $[p, q]$  和  $[pp, qq]$  不变化;

### 例 E.14 测试 find\_first\_of () 算法

```
int main ()
{
    int a [] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34};
    int b [] = {6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
    int * r = find_first_of (a, a+10, b, b+10);
    cout << " *r=" << *r << '\n';           // 这是元素 a [7]
    cout << "r-a=" << r-a << '\n';          // 这是下标 i
}

*r=8
r-a=6
```

find\_if (p, q, P ());

//返回在段  $[p, q]$  中的  $P(x)$  的开始位置;

//不变量:  $[p, q]$  不变化;

### 例 E.15 测试 find\_if () 算法

```
int main ()
{
    int a [] = {2, 4, 8, 16, 32, 64, 128, 256, 512};
    int * r = find_if (a, a+10, Odd ());
    cout << " *r=" << *r << '\n';           // 这是元素 a [1]
    cout << "r-a=" << r-a << '\n';          // 这是下标 i
    r = find_if (a, a+5, Odd ());
}
```

```
cout << "r=" << r << ", r-a=" << r-a << "\n"; // 这是元素 a[i]
cout << "r-a=" << r-a << "\n"; // 这是下标 i

*r=333
r-a=8
*r=64
r-a=5
```

```
for_each(p, q, f);
//把函数 f(x) 应用于段 [p, q] 中的每个元素;
```

### 例 E.16 测试 for\_each() 算法

```
int main()
{ int a[10] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34};
  for_each(a, a+10, print);
}
0 1 1 2 3 5 8 13 21 34
```

```
generate(p, q, f);
//把对 f(x) 连续调用的输出结果赋值给 [p, q]
```

### 例 E.17 测试 generate() 算法

```
long fibonacci();

int main()
{ int a[10] = {0};
  generate(a, a+10, fibonacci);
  print(a, 10);
}

long fibonacci()
{ static int f1=0, f2=1;
  int t0=f1;
  f1 = f2;
  f2 += t0;
  return f0;
}
n=10: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

```
generate_n(p, q, pp, qq);
//把对 f(x) 连续调用的输出结果赋值给 [p, q+pp-qq]
```

### 例 E.18 测试 generate\_n() 算法

```
long fibonacci();
```

```
int main ()
{ int a [10] = {0};
  generate_n (a, 10, fibonacci);
  print (a, 10);
}

long fibonacci ()
{ static int f1 = 0, f2 = 1;
  int f0 = f1;
  f1 = f2;
  f2 += f0;
  return f0;
}
n=10: {0, 1, 1, 2, 3, 5, 8, 13, 21, 34}
```

**includes (p, q, pp, qq):**  
 //当且仅当 [pp, qq] 中的每个元素也在 [p, q] 中时, 返回 true  
 //前提条件: 两个段都必须是排序的  
 //不变量: [p, q] 和 [pp, qq] 不变化;

### 例 E.19 测试 includes () 算法

```
int main ()
{ int a [] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34};
  int b [] = {0, 1, 2, 3, 4};
  bool found = includes (a, a+10, b, b+5);
  cout << "found=" << found << '\n';
  found = includes (a, a+10, b, b+4);
  cout << "found=" << found << '\n';

  found = 0
  found = 1
```

**inner\_product (p, q, pp, x)**  
 //返回 x 和 [p, q] 与 [pp, pp+n] 内积的和, 此时 n = q - p;  
 //不变量: [p, q] 和 [pp, qq] 不变化;

### 例 E.20 测试 inner\_product () 算法

```
int main ()
{ int a [] = {1, 3, 5, 7, 9};
  int b [] = {4, 3, 2, 1, 0};
  int dot = inner_product (a, a+4, b, 1000);
  cout << "dot=" << dot << '\n';

  sum = 1030
```

```
inplace_merge(p, r, q);
//合并段 [p, r[ 和段 [r, q[;
//前提条件: 这两个段必须是邻近的和排序的;
//后续条件: 段 [p, r[ 是排序的;
```

### 例 E.21 测试 inplace\_merge () 算法

```
int main ()
{ int a[] = {22, 55, 66, 88, 11, 33, 44, 77, 99};
  print(a, 9);
  inplace_merge(a, a+4, a+9);
  print(a, 9);
```

```
n=9: |22, 55, 66, 88, 11, 33, 44, 77, 99|
n=9: |11, 22, 33, 44, 55, 66, 77, 88, 99|
```

```
//iter_swap(p, q):
//交换元素 *p 和 *q;
```

### 例 E.22 测试 iter\_swap () 函数

```
int main ()
{ int a[] = {11, 22, 33, 44, 55, 66, 77, 88, 99};
  int b[] = {10, 20, 30, 40, 50, 60, 70, 80, 90};
  print(a, 9);
  print(b, 9);
  iter_swap(a+4, b+7);
  print(a, 9);
  print(b, 9);
}
```

```
n=9: |11, 22, 33, 44, 55, 66, 77, 88, 99|
n=9: |10, 20, 30, 40, 50, 60, 70, 80, 90|
n=9: |11, 22, 33, 44, 50, 60, 70, 80, 90|
n=9: |10, 20, 30, 40, 50, 60, 70, 80, 90|
```

```
lexicographical_compare(p, q, pp, qq):
//按词典编纂的顺序比较两个段 [pp, qq[ 和 [p, q[;
//当且仅当第一个优先于第二个时返回 true
//不变量: [p, q[ 和 [pp, qq[ 不变化;
```

### 例 E.23 测试 lexicographical\_compare () 算法

```
void test(char*, int, char*, int);
```

```
int main ()
{ char* s1 = "COMPUTER";
  char* s2 = "COMPUTABLE";
```

```

char * s3 = "COMPUTE";
test (s1, 3, s2, 3);
test (s1, 8, s2, 10);
test (s1, 8, s3, 7);
test (s2, 10, s3, 7);
test (s1, 7, s3, 7);

```

```

char * sub (char * s, int n);

```

```

void test (char * s1, int n1, char * s2, int n2)
{
    bool lt = lexicographical_compare (s1, s1+n1, s2, s2+n2);
    bool gt = lexicographical_compare (s2, s2+n2, s1, s1+n1);
    if (lt) cout << sub (s1, n1) << " < " << sub (s2, n2) << " \n";
    else if (gt) cout << sub (s1, n1) << " > " << sub (s2, n2) << " \n";
    else cout << sub (s1, n1) << " == " << sub (s2, n2) << " \n";
}

```

```

char * sub (char * s, int n)
{
    char * buffer = new char (n+1);
    strncpy (buffer, s, n);
    buffer [n] = 0;
    return buffer;
}

```

```

COM == COM
COMPUTER > COMPUTABLE
COMPUTER > COMPUTE
COMPUTABLE < COMPUTE
COMPUTE == COMPUTE

```

```

Lower_bound (p, q, x);
// 返回 [p, q] 中 x 第一次出现的位置
// 前提条件: 这个段必须是排序的
// 不变量: [p, q] 不变化

```

### 例 E.24 测试 lower\_bound () 算法

```

int main ()
{
    int a [] = { 1, 22, 22, 33, 44, 44, 44, 55, 66 };
    int * p = lower_bound (a, a+9, 44);
    cout << " *p = " << *p << " \n";
    cout << " p-a = " << p-a << " \n";
}

```

```

 *p=44
 p-a=4

```

```

make_heap (p, q);
// 把 [p, q] 中的元素重新排列到一个堆中
// 后续条件: [p, q] 是一个堆

```

例 E.25 测试 `make_heap()` 算法

```
int main ()
{
    int a[] = {44, 88, 33, 77, 11, 99, 66, 22, 55};
    print (a, 9);
    make_heap (a, a+9);
    print (a, 9);
}

n=9: |44, 88, 33, 77, 11, 99, 66, 22, 55|
n=9: |99, 88, 66, 77, 11, 33, 44, 22, 55|
```

```
max (x, y);
//返回 x 和 y 的最大值
```

例 E.26 测试 `max()` 算法

```
int main ()
{
    cout << "max (48, 84) = " << max (48, 84) << '\n';
}

max (48, 84) = 84
```

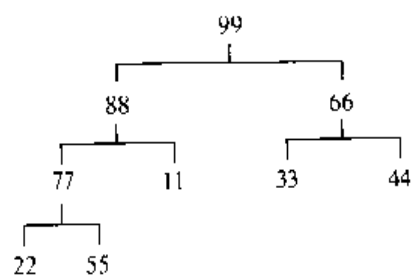


图 E.1 堆示例

```
max_element (p, q);
//返回段 [p, q] 中最大元素的位置
//不变量: [p, q] 不变化
```

例 E.27 测试 `max_element()` 算法

```
int main ()
{
    int a[] = {77, 22, 99, 55, 11, 88, 44, 33, 66};
    const int * p = max_element (a, a+9);
    cout << "*p=" << *p << '\n';
    cout << "p-a=" << p-a << '\n';
}

*p=99
p-a=2
```

```
merge (p, q, pp, qq, ppp);
//把段 [p, q] 和段 [pp, qq] 合并到段 [ppp, ppp+n] 中。
//此时 n=q-p+qq-pp;
//前提条件: [p, q] 和 [pp, qq] 必须是排序的;
//后续条件: 段 [ppp, ppp+n] 是排序的;
//不变量: [p, q] 和 [pp, qq] 是不变化的;
```

例 E.28 测试 `merge()` 函数

```
int main ()
```

```

} int a [] = {22, 55, 66, 88};
  int b [] = {11, 33, 44, 77, 99};
  int c [9];
  merge (a, a+4, b, b+5, c);
  print (c, 9);
}
// 9: 11, 22, 33, 44, 55, 66, 77, 88, 99

```

```

min (x, y);
//返回 x 和 y 的最小值

```

### 例 E.29 测试 min () 函数

```

int main ()
{ cout << "min (48, 84) =" << min (48, 84) << '\n';
}
// min (48, 84) = 48

```

```

min_element (p, q);
//返回在段 [p, q] 中的最小值元素的位置
//不变量: [p, q] 是不变化的

```

### 例 E.30 测试 min\_element () 函数

```

int main ()
{ int a [] = {77, 22, 99, 55, 11, 88, 44, 33, 66};
  const int * p = min_element (a, a+9);
  cout << "*p=" << *p << '\n';
  cout << "p-a=" << p-a << '\n';
}
// *p=11
// p-a=4

```

```

mismatch (p, q, pp);
//返回一对迭代符，它们给出在 [p, q] 和 [pp, qq] 中首先匹配
//的元素的位置
//如果这两个段完全匹配，那么返回它们的末尾
//不变量: [p, q] 和 [pp, qq] 是不变化的

```

### 例 E.31 测试 mismatch () 算法

```

int main ()
{ char * s1 = "Aphrodite, Apollo, Ares, Artemis, Athena";
  char * s2 = "Aphrodite, Apallo, Ares, Artimis, Athens";
  int n = strlen (s1);
  cout << "n=" << n << '\n';
}

```



```

pair<char*, char*> x = mismatch (s1, s1+n, s2);
char* p1 = x.first;
char* p2 = x.second;
cout << " *p1=" << *p1 << ", *p2=" << *p2 << '\n';
cout << "p1-s1=" << p1-s1 << '\n';
}
n=40
*p1=o, *p2=a
p1-s1=13

```

**next\_permutation (p, q);**

//排列 [p, q [ 中的元素; n! 个调用将会是 n 个元素的 n! 个排列的循环, 此时, n=q-p;

### 例 E.32 测试 next\_permutation () 算法

```

int main ()
{ char* s="ABCD";
  for (int i=0; i<24; i++)
    { next_permutation (s, s+4);
      cout << (i%8?'\\t':'\\n') << s;
    }
}

```

```

ABDC ACBD ACDE ADBC ADCB BACD BADC BCAD
BCDA BDAC BDCA CABD CADB CBAD CBDA CDAB
CDEA DABC DACE DBAC DECA DCAB DCBA ABCD

```

next\_permutation () 算法是 prev\_permutation () 算法的逆算法 (383 页中例 E.39)

**nth\_element (p, r, q);**

//重新排列 [p, q [ 中的元素, 使得 \*r 把它分为两个子段 [p, r1 [ 和 [r2, q [, 此时 r1 是 \*r 的新位置,

//[p, r1 [ 中的所有元素小于或等于 \*r, 并且 [r2, q [ 中的所有元素大于或等于 \*r;

//\*r 叫做枢轴元素

### 例 E.33 测试 nth\_element () 算法

```

int main ()
{ int a[] = {77, 22, 99, 55, 44, 88, 11, 33, 66};
  print (a, 9);
  nth_element (a, a+3, a+9);
  print (a, 9);
}
n=9: {77, 22, 99, 55, 44, 88, 11, 33, 66}
n=9: {11, 22, 33, 44, 55, 88, 66, 99, 77}

```

**partial\_sort (p, r, q);**

//对 [p, q [ 开始的 r-p 个元素排序, 把它们放入 [p, r [ 中,

//然后把剩余的  $q-r$  个元素移  $[r, q]$  中;

### 例 E.34 测试 `partial_sort()` 函数

```
int main ()
{
    int a [] = {77, 22, 99, 55, 44, 88, 11, 33, 66};
    print (a, 9);
    partial_sort (a, a+3, a+9);
    print (a, 9);
}
n=9: {77, 22, 99, 55, 44, 88, 11, 33, 66}
n=9: {11, 22, 33, 99, 77, 88, 55, 44, 66}
```

```
partial_sort_copy (p, q, pp, qq);
//把  $[p, q]$  中的最小的  $qq-pp$  个元素复制以排列的顺序复制到  $[pp, qq]$  中;
//然后把剩余的  $n$  个元素复制  $[qq, qq-n]$  中;
//此时,  $r = q-p-pp-qq$ ;
//不变量:  $[p, q]$  是不变化的
```

### 例 E.35 测试 `partial_sort_copy()` 算法

```
int main ()
{
    int a [] = {77, 22, 99, 55, 44, 88, 11, 33, 66};
    print (a, 9);
    int b [3];
    partial_sort_copy (a, a+9, b, b+3);
    print (a, 9);
    print (b, 3);
}
n=9: {77, 22, 99, 55, 44, 88, 11, 33, 66}
n=9: {77, 22, 99, 55, 44, 88, 11, 33, 66}
n=3: {11, 22, 33}
```

```
partial_sum (p, q, pp);
//不变量:  $a[p, q]$  是不变化的
//后续条件: 对  $[pp, pp+q-p]$  中的每个  $b[i]$  都有:  $b[i] = a[0] + \dots + a[i]$ 
```

### 例 E.36 测试 `partial_sum()` 算法

```
int main ()
{
    int a [] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34};
    int b [10];
    partial_sum (a, a+10, b);
    print (a, 10);
    print (b, 10);
}
```

```
n=10: {0, 1, 1, 2, 3, 5, 8, 13, 21, 34}
n=10: {0, 1, 2, 4, 7, 12, 20, 33, 54, 88}
```

`partial_sum()` 算法是 `adjacent_difference()` 算法的逆算法 (370 页中的例 E.2)

```
partition(p, q, P());
//把 [p, q] 划分为 [p, r] 和 [r, q] 使得
//当且仅当 P(x) 是 true 时, x 在 [p, r] 中
```

### 例 E.37 测试 `partition()` 算法

```
int main()
{
    int a[] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34};
    print(a, 10);
    partition(a, a+10, Odd());
    print(a, 10);
}
n=10: {0, 1, 1, 2, 3, 5, 8, 13, 21, 34}
n=10: {21, 1, 1, 13, 3, 5, 8, 2, 0, 34}
```

```
pop_heap(p, q);
//把 *p 移入 temp 中, 然后把元素移动到左边, 使得剩余的元素把 [p, q-1] 中的一个堆
//形成一个堆, 然后把 temp 复制到 (q-1) 中
//前提条件: [p, q] 必须是一个堆
//后续条件: [p, q-1] 是一个堆
```

### 例 E.38 测试 `pop_heap()` 算法 (见图 E.2 所示)

```
int main()
{
    int a[] = {44, 88, 33, 77, 11, 99, 66, 22, 55};
    print(a, 9);
    make_heap(a, a+9);
    print(a, 9);
    pop_heap(a, a+9);
    print(a, 9);
    print(a, 8);
}
n=9: {44, 88, 33, 77, 11, 99, 66, 22, 55}
n=9: {99, 88, 66, 77, 11, 33, 44, 22, 55}
n=9: {88, 77, 66, 55, 11, 33, 44, 22, 99}
n=8: {88, 77, 66, 55, 11, 33, 44, 22}
```

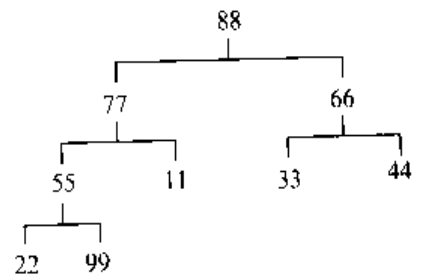


图 E.2 弹出堆示例

```
prev_permutation(p, q);
//排列 [p, q] 中的元素; n! 个调用将会是 n 个元素的 n! 个排列的向后循环, 此时, n=q-p;
```

### 例 E.39 测试划分 `prev_permutation()` 算法

```
int main()
```

```

| char * s = "ABCD";
| for (int i = 0; i < 24; i++)
| | prev_permutation(s, s + 4);
| | cout << (i % 8 ? '\t' : '\n') << s;
|
|
|

```

```

DCBA DCAB DBCA DBAC DACB DABC CDBA CDAB
CBDA CBAD CADB CABD EDCA BDAC ECDA BCAD
BADC BADC ADCB ADBC ACDB ACBD ABDC ABCD

```

```

push_heap(p, q);
//把在 * (q-1) 的元素加到 [p, q-1] 中, 使得 [p, q] 是一个堆
//前提条件: [p, q-1] 必须是一个堆
//后续条件: [p, q] 是一个堆

```

#### 例 E.40 测试 push\_heap() 算法 (见图 E.3)

```

int main ()
| int a [] = {66, 44, 88, 33, 55, 11, 99, 22, 77};
| print(a, 8);
| make_heap(a, a + 8);
| print(a, 8);
| print(a, 9);
| push_heap(a, a + 9);
| print(a, 9);
|

```

```

n=8: {66, 44, 88, 33, 55, 11, 99, 22}
n=8: {99, 55, 88, 33, 44, 11, 66, 22}
n=9: {99, 55, 88, 33, 44, 11, 66, 22, 77}
n=9: {99, 77, 88, 55, 44, 11, 66, 22, 33}

```

push\_heap() 算法逆序了 pop\_heap() 的结果。(见例 E.38)

```

random_shuffle(p, q);
//对 [pp, qq] 执行一个随机 (不是决定的) 移动

```

#### 例 E.41 测试 random\_shuffle() 算法

```

int main ()
| char * s = "ABCDEFGHJ";
| cout << s << '\n';
| for (int i = 0; i < 4; i++)
| | random_shuffle(s, s + 10);
| | cout << s << '\n';
|
|
|

```

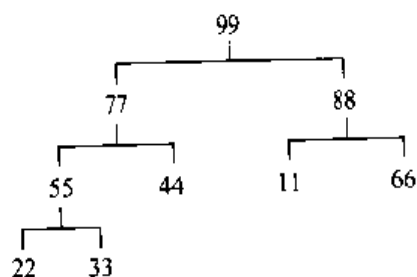


图 E.3 加入堆示例

```

ABCDEFGHIJ
CJUDBEAHGF
CFBDEIGAHJ
IDJABDEGHC
DEJIFEGACH

```

```
remove(p, q, x);
```

//从 [p, q] 中删除所有出现 x 的地方, 移动 (复制) 剩余的元素到左边;

//不变量: 段的长度保持不变

### 例 E.42 测试 remove () 函数

```
int main ()
```

```
{ char* s = "All is flux, nothing is stationary.";
```

```
  int l = strlen(s);
```

```
  int n = count(s, s+1, ' ');
```

```
  cout << "l=" << l << '\n';
```

```
  cout << "n=" << n << '\n';
```

```
  remove(s, s+1, ' ');
```

```
  cout << s << '\n';
```

```
  s[l-n] = 0; // 截去 s
```

```
  cout << s << '\n';
```

```
{
```

```
l=35
```

```
n=5
```

```
Allisflux, nothingisstationaryonary.
```

```
Allisflux, nothingisstationary.
```

由于删除了 5 个空格, 剩余的 5 个字母在它们的复制品被移到左边后保留。

```
remove_copy(p, q, pp, x);
```

//复制 [p, q] 中所有和 [pp, pp+n] 中的 x 不匹配的元素,

//此时 n 是不匹配元素的数目

//返回 pp+n;

//不变量: [p, q] 保持不变;

### 例 E.43 测试 remove\_copy () 算法

```
int main ()
```

```
{ char* s = "All is flux, nothing is stationary.";
```

```
  char buffer[80];
```

```
  int l = strlen(s);
```

```
  int n = count(s, s+1, ' ');
```

```
  cout << "l=" << l << '\n';
```

```
  cout << "n=" << n << '\n';
```

```
  char* ss = remove_copy(s, s+1, buffer, ' ');
```

```
  *ss = 0; //去除缓冲区
```

```
  cout << s << '\n';
```

```
  cout << buffer << '\n';
```

```
  cout << ss - buffer << '\n';
```

```

1=35
n=5
Allisflux, nothing is stationary.
Allisflux, nothingisstationary.
30

```

```

remove_copy_if (p, q, pp, P ());
//对每个! P (x) 复制 [p, q [中的所有元素到 [pp, pp+n [中,
//此时 n 是不匹配的元素数目;
//返回 pp+n;
//不变量: [p, q [保持不变;

```

#### 例 E.44 测试 remove\_copy\_if () 算法

```

class Blank
{ public:
    bool operator () (char c) { return c == ' '; }
};

int main ()
{ char * s = "All is flux, nothing is stationary.";
  char buffer [80];
  int l = strlen (s);
  int n = count (s, s+l, ' ');
  cout << "l = " << l << '\n';
  cout << "n = " << n << '\n';
  char * ss = remove_copy_if (s, s+l, buffer, Blank ());
  *ss = 0; // 去除缓冲区
  cout << s << '\n';
  cout << buffer << '\n';
  cout << ss - buffer << '\n';
}

```

```

1=35
n=5
All is flux, nothing is stationary.
Allisflux, nothingisstationary.
30

```

除了使用了谓词外，这和例 E.43 完全相同。

```

remove_if (p, q, P ());
//对每个! P (x) 复制 [p, q [中的所有元素,
//移动 (复制) 剩余的元素到左边;

```

#### 例 E.45 测试 remove\_if () 函数

```

class Blank
{ public:
    bool operator () (char c) { return c == ' '; }
};

```

```
};

int main ()
{ char * s="All is flux, nothing is stationary.";
  int l = strlen (s);
  int n = count (s, s+1, ' ');
  cout << "l=" << l << '\n';
  cout << "n=" << n << '\n';
  remove_if (s, s+1, Blank ());
  cout << s << '\n';
  s[l-n] = 0;
  cout << s << '\n';
}
```

```
l=35
```

```
n=5
```

```
Allisflux, nothingisstationaryonary.
```

```
Allisflux, nothingisstationary.
```

除了使用了谓词外，这和例 E.42 完全相同。

**replace (p, q, x, y);** //用 [p, q[ 中的 y 替代所有出现 x 的地方  
//不变量：段的长度保持不变；

#### 例 E.46 测试 replace () 算法

```
int main ()
{ char * s="All is flux, nothing is stationary.";
  int l = strlen (s);
  cout << s << '\n';
  replace (s, s+1, ' ', '!');
  cout << s << '\n';
}
```

```
All is flux, nothing is stationary.
```

```
All! is! flux,! nothing! is! stationary.
```

**replace\_copy (p, q, pp, x, y);**  
//把 [p, q[ 中的所有元素复制到 [pp, pp+n[ 中，用 y 替代每处出现 x 的地方，此时 n = q - p;  
//返回 pp+n;  
//不变量：[p, q[ 保持不变；

#### 例 E.47 测试 replace\_copy () 算法

```
int main ()
{ char * s="All is flux, nothing is stationary.";
  cout << s << '\n';
  int l = strlen (s);
  char buffer [80];
  char * ss = replace_copy (s, s+1, buffer, 'n', 'N');
  *ss = 0; // 去除打印的缓冲区
  cout << s << '\n';
}
```

```
cout << buffer << '\n';
|
All is flux, nothing is stationary.
All is flux, nothing is stationary.
All is flux, nothing is stationary.
```

```
replace_copy_if (p, q, pp, P (), y);
//把 [p, q [ 中的所有元素复制到 [pp, pp+n [ 中, 用 y 替代 P (x) 中的每个 x,
//此时 n = q-p;
//返回 pp+n,
//不变量: [p, q [ 保持不变;
```

### 例 E.48 测试 replace\_copy\_if () 算法

```
class Blank
{ public:
    bool operator () (char c) { return c == ' '; }
};

int main ()
{ char* s = "All is flux, nothing is stationary.";
  int l = strlen (s);
  char buffer [80];
  cout << s << '\n';
  char* ss = replace_copy_if (s, s+l, buffer, Blank (), '!');
  *ss = 0; // 去除缓冲区
  cout << s << '\n';
  cout << buffer << '\n';
|
All is flux, nothing is stationary.
All is flux, nothing is stationary.
All! is! flux,! nothing! is! stationary.
```

除了使用了谓词外, 这和例 E.47 完全相同。

```
replace_if (p, q, P (), y);
//用 [p, q [ 中的 y 替代 P (x) 中的每个 x;
```

### 例 E.49 测试 replace\_if () 算法

```
class Blank
{ public:
    bool operator () (char c) { return c == ' '; }
};

int main ()
{ char* s = "All is flux, nothing is stationary.";
  int l = strlen (s);
  cout << s << '\n';
  replace_if (s, s+l, Blank (), '!');
```



```
cout << s << '\n';
}
All is flux, nothing is stationary.
All! is! flux,! nothing! is! stationary.
```

```
reverse (p, q);
//逆序段 [p, q[;
```

### 例 E.50 测试 reverse () 函数

```
int main ()
{ char * s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  cout << s << '\n';
  reverse (s, s+26);
  cout << s << '\n';
}
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ZYXWVUTSRQPONMLKJIHGFEDCBA
```

```
reverse_copy (p, q, pp);
//以相反的顺序把段 [p, q[复制到 [pp, pp+n[中
//此时 n = q-p;
//返回 pp+n;
//不变量: [p, q[保持不变;
```

### 例 E.51 测试 reverse\_copy () 算法

```
int main ()
{ char * s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  cout << s << '\n';
  char buffer [80];
  char * ss = reverse_copy (s, s+26, buffer);
  *ss = 0; // 去除打印的缓冲区
  cout << s << '\n';
  cout << buffer << '\n';
}
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ZYXWVUTSRQPONMLKJIHGFEDCBA
```

```
rotate (p, r, q);
//把 [r, q[向左移动 r 个位置到 [p, p+q-r[中,
//并且把 [p, r[中靠右的末尾部分移到 [p+q-r, q[中;
```

### 例 E.52 测试 rotate () 算法

```
int main ()
```

```
char * s = "ABCDEFGHILJKLMNOPQRSTUVWXYZ";
cout << s << '\n';
rotate (s, s+4, s+26);
cout << s << '\n';
```

```
ABCDEFGHILJKLMNOPQRSTUVWXYZ
EFGHILJKLMNOPQRSTUVWXYZABCD
```

```
rotate_copy (p, r, q, pp):
//复制段 [p, q]到段 [pp, pp+m]中, 此时 m = q - r,
//并且复制段 [p, r]到 [pp+n, pp+n]中, 此时 n = q - p;
//返回 pp+n+n;
//不变量: [p, q]保持不变;
```

### 例 E.53 测试 rotate\_copy () 算法

```
int main ()
{ char * s = "ABCDEFGHILJKLMNOPQRSTUVWXYZ";
  cout << s << '\n';
  char buffer [80];
  char * ss = rotate_copy (s, s+4, s+26, buffer);
  *ss = 0; // 去除打印的缓冲区
  cout << s << '\n';
  cout << buffer << '\n';
}
```

```
ABCDEFGHILJKLMNOPQRSTUVWXYZ
ABCDEFGHILJKLMNOPQRSTUVWXYZ
EFGHILJKLMNOPQRSTUVWXYZABCD
```

```
search (p, q, pp, qq):
//在 [p, q]中查找子列 [pp, qq];
//如果找到, 返回它第一次出现的位置 r;
//否则, 返回 q;
//后续条件: 或者 r = q 或者 [1, r-n] = [pp, qq], 此时 r = qq - pp;
//不变量: [p, q]保持不变;
```

### 例 E.54 测试 search () 算法

```
int main ()
{ char * p = "ABCDEFGHILJK ABCDEFGHILJK!";
  char * pp = "HIJK";
  char * r = search (p, p+24, pp, pp-4);
  int n = r - p; // p 中子串前的字符数
  cout << "n = r - p = " << n << '\n';
  cout << " * r = " << *r << '\n';
  cout << p << '\n';
  cout << string (n, ' ') << pp << string (20 - n, ' ') << '\n';
  pp = "LMNOP";
```

```

    r = search (p, p+24, pp, pp+5);
    n = r-p;
    cout << "n=r-p=" << n << '\n';
    cout << p << '\n';
    cout << string (n, '-') << '\n';
}

```

```

n=r-p=7
*r=H
ABCDEFGHIJKLABCDEFGHIJKL
-----HIJK-----
n=r-p=24
ABCDEFGHIJKLABCDEFGHIJKL
-----

```

```

search_n (p, q, n, x);
//查找在 [p, q] 中 x 的 n 个连续的复制的子列;
//如果找到, 返回它第一次出现的位置 r;
//否则, 返回 q;
//后续条件: 或者 r = q 或者 [r, r+n] = [pp, qq], 此时 n = qq-pp;
//不变量: [p, q] 保持不变;

```

### 例 E.55 测试 search\_n () 算法

```

int main ()
{
    char * p = "0010111001111110";
    char * r = search_n (p, p+16, 3, '1');
    int m = r - p; // p 中子串前的字符数
    cout << "m=r-p=" << m << '\n';
    cout << p << '\n';
    cout << string (m, '-') << string (3, '1') << string (13-m, '-') << '\n';
    r = search_n (p, p+16, 4, '1');
    m = r - p; // p 中子串前的字符数
    cout << "m=r-p=" << m << '\n';
    cout << p << '\n';
    cout << string (m, '-') << string (4, '1') << string (12-m, '-') << '\n';
}

```

```

m=r-p=4
0010111001111110
----111-----
m=r-p=9
0010111001111110
-----1111--

```

```

set_difference (p, q, pp, qq, ppp);
//把在 [p, q] 而不在 [pp, qq] 中的元素复制到 [ppp, ppp+n] 中;
//返回 ppp+n, 此时 n 是复制的元素数目
//不变量: [p, q] 和 [pp, qq] 保持不变;

```

**例 E.56 测试 set\_difference () 算法**

```
int main ()
{
    char * p = "ABCDEFGHILJ";
    char * pp = "AEIOUXYZ";
    char ppp [16];
    char * qq = set_difference (p, p+10, pp, pp+8, ppp);
    cout << p << '\n';
    cout << pp << '\n';
    *qq = 0; // 结束 ppp 串
    cout << ppp << '\n';
}
```

```
ABCDEFGHILJ
AEIOUXYZ
BCDEFGHI
```

```
set_intersection (p, q, pp, qq, ppp);
//把在 [p, q [也在 [pp, qq [中的元素复制到 [ppp, ppp+n [中;
//返回 ppp+n, 此时 n 是复制的元素数目
//不变量: [p, q [和 [pp, qq [保持不变;
```

**例 E.57 测试 set\_intersection () 算法**

```
int main ()
{
    char * p = "ABCDEFGHILJ";
    char * pp = "AEIOUXYZ";
    char ppp [16];
    char * r = set_intersection (p, p+10, pp, pp+8, ppp);
    cout << p << '\n';
    cout << pp << '\n';
    *r = 0; // 结束 ppp 串
    cout << ppp << '\n';
}
```

```
ABCDEFGHILJ
AEIOUXYZ
AEI
```

```
set_symmetric_difference (p, q, pp, qq, ppp);
//把在 [p, q [而不在 [pp, qq [中的元素复制到 [ppp, ppp+n [中;
//并且把在 [pp, qq [而不在 [p, q [中的元素复制到 [ppp, ppp+n [中;
//返回 ppp+n, 此时 n 是复制的元素数目
//不变量: [p, q [和 [pp, qq [保持不变;
```

**例 E.58 测试 set\_symmetric\_difference () 算法**

```
int main ()
```

```

char * p = "ABCDEFGHILJ";
char * pp = "AEIOUXYZ";
char ppp[16];
char * qq = set_symmetric_difference(p, p+10, pp, pp+8, ppp);
cout << p << '\n';
cout << pp << '\n';
*qq = 0; // 结束 ppp 串
cout << ppp << '\n';

```

```

ABCDEFGHILJ
AEIOUXYZ
BCDFGHJOUXYZ

```

**set\_union** (p, q, pp, qq, ppp);  
 //把所有在 [p, q] 和所有在 [pp, qq] 中没有复制品的元素复制到 [ppp, ppp+n] 中;  
 //返回 ppp+n, 此时 n 是复制的元素数目  
 //不变量: [p, q] 和 [pp, qq] 保持不变;

### 例 E.59 测试 set\_union () 算法

```

int main ()
{
  char * p = "ABCDEFGHILJ";
  char * pp = "AEIOUXYZ";
  char ppp[16];
  char * r = set_union(p, p+10, pp, pp+8, ppp);
  cout << p << '\n';
  cout << pp << '\n';
  *r = 0; // 终止 ppp 串
  cout << ppp << '\n';
}

```

```

ABCDEFGHILJ
AEIOUXYZ
ABCDEFGHILJOUXYZ

```

**sort** (p, q);  
 //排序 [p, q];

### 例 E.60 测试 sort () 算法

```

int main ()
{
  char * p = "GAJEHCHDIEFAGDHC";
  cout << p << '\n';
  sort(p, p+16);
  cout << p << '\n';
}

```

```

GAJEHCHDIEFAGDHC
AABCCDDEFGGHHILJ

```

```
sort_heap ();
//排序 [p, q[
```

### 例 E.61 测试 sort\_heap () 算法

```
int main ()
{ int a[] = {66, 88, 44, 77, 33, 55, 11, 99, 22};
  print (a, 9);
  make_heap (a, a+9);
  print (a, 9);
  sort_heap (a, a+9);
  print (a, 9);
}
```

```
n=9: {66, 88, 44, 77, 33, 55, 11, 99, 22}
n=9: {99, 88, 55, 77, 33, 44, 11, 66, 22}
n=9: {11, 22, 33, 44, 55, 66, 77, 88, 99}
```

```
swap (x, y):
//交换两个元素 x 和 y
```

### 例 E.62 测试 swap () 算法

```
int main ()
{ char * p = "ABCDEFGHILJ";
  cout << p << '\n';
  swap (p[2], p[8]);
  cout << p << '\n';
}
```

```
ABCDEFGHILJ
ABIDEFGHCU
```

```
transform (p, q, pp, f):
//对 [p, q[ 中的每个 x 应用函数 f (x) , 并且把结果复制到 [pp, pp+n [ 中, 此时 n=q-p;
//不变量: [p, q[ 保持不变;
```

### 例 E.63 测试 transform () 算法

```
char capital (char);

int main ()
{ char * s = "All is flux, nothing is stationary.";
  int len = strlen (s);
  char buffer [80];
  char * ss = transform (s, s+len, buffer, capital);
  *ss = 0; // 去除缓冲区
  cout << s << '\n';
  cout << buffer << '\n';
}
```

```
char capital (char c)
    return (isalpha (c) ? toupper (c) : c);
```

```
All is flux, nothing is stationary.
All IS FLUX, NOTHING IS STATIONARY.
```

```
unique (p, q):
//删除 [p, q) 中所有邻近的复制元素，向左移动它们的下标；
//返回紧凑最后移动的元素的位置
```

### 例 E.64 测试 unique () 算法

```
int main ()
{ char * s = "All is flux, nothing is stationary.";
  int len = strlen (s);
  cout << s << '\n';
  sort (s, s + len);
  cout << s << '\n';
  char * ss = unique (s, s + len);
  cout << s << '\n';
  *ss = 0; // 去除缓冲区
  cout << s << '\n';
}

All is flux, nothing is stationary.
. . .Aaafghiiiiiiillrrnnoorssstttuxy
. . .Aafghilnorstuxyillrrnnoorssstttuxy
. . .Aafghilnorstuxy
```

```
unique_copy (p, q, pp);
//把 [p, q) 中没有复制的元素复制到 [pp, pp+n) 中，
//此时 r 是 [p, q) 中独一无二的元素的数据；
//返回 pp+n；
//不变量：[p, q) 保持不变；
```

### 例 E.65 测试 unique\_copy () 算法

```
int main ()
{ char * s = "All is flux, nothing is stationary.";
  int len = strlen (s);
  cout << s << '\n';
  sort (s, s + len);
  cout << s << '\n';
  char buffer [80];
  char * ss = unique_copy (s, s + len, buffer);
  *ss = 0; // 去除打印缓冲区
  cout << s << '\n';
  cout << buffer << '\n';
}
```

```

}
All is flux, nothing is stationary.
    , .Aaafghiiiiilllnnoorssstttuxy
    , .Aaafghiiiiilllnnoorssstttuxy
    , .Aafghilnorstuxy

```

```

upper_bound(p, q, x);
//返回紧随 [pp, qq[ 中的 x 的最后事件的位置
//前提条件: [p, q[ 必须是排序的
//不变量: [p, q[ 未变化

```

### 例 E.66 测试 upper\_bound() 算法

```

int main ()
{
    int a [] = {11, 22, 22, 33, 44, 44, 44, 55, 66};
    int * p = upper_bound(a, a+9, 44);
    cout << " *p=" << *p << '\n';
    cout << "p-a=" << p-a << '\n';
}

```

```

 *p=55
p-a=7

```



## 附录 F 标准 C 库

这个附录描述了标准 C 库中提供的预定义函数。每个条目列出了函数名、它的原型、一个它所做什么的简单描述和声明出处的头文件。

| 函数名         | 原型和描述                                                                                                                                                                                                        | 头文件          |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|
| abort ()    | void abort ();<br>中止程序                                                                                                                                                                                       | < cstdlib >  |
| abs () int  | abs (int n);<br>返回 n 的绝对值                                                                                                                                                                                    | < cstdlib >  |
| acos ()     | double acos (double x);<br>返回 x 的反余弦 (arcsine)                                                                                                                                                               | < cmath >    |
| asin ()     | double asin (double x);<br>返回 x 的正弦 (arcsine)                                                                                                                                                                | < cmath >    |
| atan ()     | double atan (double x);<br>返回 x 的反正切 (arctangent)                                                                                                                                                            | < cmath >    |
| atof ()     | double atof (const char * s);<br>返回字符串 s 中的字面上的数目                                                                                                                                                            | < cstdlib >  |
| atoi ()     | int atoi (const char * s);<br>返回字符串 s 中的字面上的数目                                                                                                                                                               | < cstdlib >  |
| atol ()     | long atol (const char * s);<br>返回字符串 s 中的字面上的数目                                                                                                                                                              | < cstdlib >  |
| bad ()      | int ios:: bad ();<br>如果设置了 badbit 返回非零；否则返回 0                                                                                                                                                                | < iostream > |
| bsearch ()  | void * bsearch (const void * x, void * a, size_t n, size_t s, int (* cmp) (const void, const void *));<br>实现二分法查找算法来查找已排序的 n 个元素的数组 a 中的 x，每个元素的大小是 s，使用函数 * cmp 来比较任何两个这样的元素。如果找到，返回一个指向这个元素的指针，如果未找到，返回空指针 | < cstdlib >  |
| ceil ()     | double ceil (double x);<br>返回接近                                                                                                                                                                              | < cmath >    |
| clear ()    | void ios:: clear (int n=0);<br>将流的状态设为 n                                                                                                                                                                     | < iostream > |
| clearerr () | void clearerr (FILE * p);<br>对文件 * p 清除文件结束标记和错误标记                                                                                                                                                           | < cstdio >   |

(续表)

| 函数名         | 原型和描述                                                                                                                                                | 头文件          |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|
| close ()    | void fstreambase:: close ();<br>关闭和宿主对象相关的文件                                                                                                         | < iostream > |
| cos ()      | double cos (double x);<br>返回 x 的余弦                                                                                                                   | < cmath >    |
| cosh ()     | double cosh (double x);<br>返回 x 的双曲线余弦: $(e^x + e^{-x}) / 2$ 。                                                                                       | < cmath >    |
| difftime () | Double difftime (time_t t1, time_t t0);<br>返回从时间 t0 到 t1 的时间差 (以秒计算)                                                                                 | < ctime >    |
| eof ()      | int ios:: eof ();<br>如果是最后一位返回非零; 否则返回 0;                                                                                                            | < iostream > |
| exit ()     | void exit (int n);<br>结束程序, 对正调用的进程返回 n                                                                                                              | < cstdlib >  |
| exp ()      | double exp (double x);<br>返回 x 的指数; $e^x$                                                                                                            | < cmath >    |
| fabs ()     | double fabs (double x);<br>返回 x 的绝对值                                                                                                                 | < cmath >    |
| fail ()     | int ios:: fail ();<br>如果设置 failbit 位, 则返回非零, 否则返回 0                                                                                                  | < iostream > |
| fclose ()   | int fclose (FILE* p);<br>关闭文件 *p, 并且刷新缓冲区。如果成功, 返回 0, 否则返回 EOF                                                                                       | < cstdio >   |
| fgetc ()    | int fgetc (FILE* p);<br>如果可能, 读入并返回文件 *p 中的下一个字符, 否则返回 EOF                                                                                           | < cstdio >   |
| fgets ()    | char* fgets (char* s, int n, FILE* p);<br>从文件 *p 中返回下一行, 并存放在 *s 中。“下一行”指或者是下 n-1 个字符, 或者是下一行结束符前的所有字符, 先满足条件者。s 中的字符后会自动加 NUL。如果成功, 返回 s, 否则返回 NULL | < cstdio >   |
| fill ()     | char ios:: fill ();<br>返回当前的填充字符<br>char ios:: fill (char c);<br>将填充字符改为 c, 并且返回之前的填充字符                                                              | < iostream > |
| flags ()    | long ios:: flags ();<br>返回当前的格式标记<br>long ios:: flags (long n);<br>将当前的格式标记改为 n 并返回之前的格式标记                                                           | < iostream > |
| floor ()    | double floor (double x);<br>返回小于 x 的最大整数                                                                                                             | < cmath >    |
| flush ()    | ostream& ostream:: flush ();<br>刷新输出流并返回更新后的流                                                                                                        | < iostream > |

(续表)

| 函数名        | 原型和描述                                                                                                                                                                                                                                                                                                                | 头文件        |
|------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------|
| fopen ()   | FILE * fopen (const char * p, const char * s);<br>打开文件 * p, 如果成功返回表示此文件的数据结构指针, 否则返回 NULL。字符串 s 表示文件的方式: “r” 表示读, “w” 表示写, “a” 表示添加, “r+” 表示读写, “w+” 表示读写, “a+” 表示读、添加文件                                                                                                                                             | <stdio>    |
| fprintf () | int fprintf (FILE * p, const char * s, ...);<br>向文件 * p 中写入格式化输出。如果成功, 返回写入的字符数, 否则返回负数                                                                                                                                                                                                                              | <stdio>    |
| fputc ()   | int fputc (int c, FILE * p);<br>向文件 * p 中写入字符 c。返回写入的字符或不成功里写入 EOF                                                                                                                                                                                                                                                   | <stdio>    |
| fputs ()   | int fputs (const char * s, FILE * p);<br>向文件 * p 中写入串 s。如果成功, 返回非负数, 否则返回 EOF                                                                                                                                                                                                                                        | <stdio>    |
| fread ()   | size_t fread (void * s, size_t k, size_t n, FILE * p)<br>读入并返回 n 个大小为 k 的项, 并存放在内存中的 s 处。返回读入的个数                                                                                                                                                                                                                     | <stdio>    |
| fscanf ()  | int fscanf (FILE * p, const char * s, ...);<br>从文件 * p 中读入格式化输入, 并放在内存中的 s 处。如果到文件尾, 返回 EOF, 否则返回读入的内存的项数                                                                                                                                                                                                            | <stdio>    |
| fseek ()   | int fseek (FILE * p, long k, int base);<br>将文件 * p 的位置标记重新定在从 base 开始 k 字节处。base 必须是表示文件开头的 SEEK_SET, 表示当前位置的 SEEK_CUR 及表示文件尾的 SEEK_END                                                                                                                                                                              | <stdio>    |
| ftell      | long ftell (FILE * p);<br>返回文件 * p 的位置标记或 -1                                                                                                                                                                                                                                                                         | <stdio>    |
| fwrite ()  | size_t fwrite (void * s, size_t k, size_t n, FILE * p)<br>向文件中写入 n 个大小为 k 的项, 并返回写入的数目                                                                                                                                                                                                                               | <stdio>    |
| gcount ()  | int istream::gcount ();<br>返回最近一次读入的字符数                                                                                                                                                                                                                                                                              | <iostream> |
| get ()     | int istream::get ();<br>istream& istream::get (signed char& c);<br>istream& istream::get (unsigned char& c);<br>istream& istream::get (signed char * b, int n, char e = '\n');<br>istream& istream::get (unsigned char * b, int n, char e = '\n');<br>从流 istream 中读入下一个字符, 第一个返回 c 或都 EOF。后两个读入最多 n 个字符至 b 中, 直到遇到 e | <iostream> |
| getc ()    | int getc (FILE * p);<br>除了用宏实现外, 和 get 一样                                                                                                                                                                                                                                                                            | <stdio>    |
| getchar () | int getchar ();<br>返回标准输入中的下一个字符, 或返回 EOF                                                                                                                                                                                                                                                                            | <stdio>    |

(续表)

| 函数名         | 原型和描述                                                                                                                               | 头文件        |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------|------------|
| gets ()     | char* gets (char* s);<br>从标准输入中读入下一行, 并存放在 s 中, 返回 s 或者如果没有字符读入, 返回 NULL.                                                           | <stdio>    |
| good ()     | int ios:: good ();<br>如果流的状态是 0, 返回非零, 否则返回 0                                                                                       | <iostream> |
| ignore ()   | istream& ignore (int n = 1, int e = EOF);<br>从流中取出 n 个字符, 或遇到 e 返回流                                                                 | <iostream> |
| isalnum ()  | int isalnum (int c);<br>如果字符 c 是字母或数字, 返回非零, 否则返回 0                                                                                 | <cctype>   |
| isalpha ()  | int isalpha (int c);<br>如果字符 c 是字母, 返回非零, 否则返回 0                                                                                    | <cctype>   |
| isctrl ()   | int isctrl (int c);<br>如果字符 c 是控制字符, 返回非零, 否则返回 0                                                                                   | <cctype>   |
| isdigit ()  | int isdigit (int c);<br>如果字符 c 是数字, 返回非零, 否则返回 0                                                                                    | <cctype>   |
| isgraph ()  | int isgraph (int c);<br>如果字符 c 是非空可打印字符, 返回非零, 否则返回 0                                                                               | <cctype>   |
| islower ()  | int islower (int c);<br>如果字符 c 是小写字母, 返回非零, 否则返回 0                                                                                  | <cctype>   |
| isprint ()  | int isprint (int c);<br>如果字符 c 是可打印字符, 返回非零, 否则返回 0                                                                                 | <cctype>   |
| ispunct ()  | int ispunct (int c);<br>如果字符 c 是标点符, 除了字母、数字及空格, 返回非零, 否则返回 0                                                                       | <cctype>   |
| isspace ()  | int isspace (int c);<br>如果字符 c 是空白字符, 包括空格 ' ', 换页 '\f', 换行符 '\n', 回车符 '\r', 横向制表符 '\t' 及纵向制表符 '\v', 返回非零, 否则返回 0                   | <cctype>   |
| isupper ()  | int isupper (int c);<br>如果字符 c 是大写字母, 返回非零, 否则返回 0                                                                                  | <cctype>   |
| isxdigit () | int isxdigit (int c);<br>如果字符 c 是 10 个数字或 12 个 16 进制字符: 'a', 'b', 'c', 'd', 'e', 'f', 'A', 'B', 'C', 'D', 'E', 'F' 之一, 返回非零, 否则返回 0 | <cctype>   |
| labs ()     | long labs (long n);<br>返回 n 的绝对值                                                                                                    | <stdlib>   |
| log ()      | double log (double x);<br>返回 x 的以 e 为底的自然对数                                                                                         | <cmath>    |
| log10 ()    | double log10 (double x);<br>返回 x 的以 10 为底的普通对数                                                                                      | <cmath>    |

(续表)

| 函数名          | 原型和描述                                                                                                                                                                                                                                                                 | 头文件         |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|
| memchr ()    | void* memchr (const void* s, int c, size_t k);<br>在从地址 s 开始的 k 字节内存中, 查找字符 c。如果发现, 返回第一个发现的地址, 否则返回 NULL。                                                                                                                                                             | < string >  |
| memcmp ()    | int memcmp (const void* s1, const void* s2, size_t k);<br>比较以 s1 和 s2 开始的 k 字节内容, 返回负数、0 和正数, 表示根据字典顺序的小于、等于及大于。                                                                                                                                                      | < string >  |
| memcpy ()    | void* memcpy (const void* s1, const void* s2, size_t k);<br>将 s2 开始的 k 字节内容复制到 s1 开始处, 返回 s1。                                                                                                                                                                         | < string >  |
| memmove ()   | int memmove (const void* s1, const void* s2, size_t k);<br>除了串可以重叠, 与 memcpy 一样。                                                                                                                                                                                      | < string >  |
| open ()      | void fstream::open (const char* f, int m, int p = filebuf::openprot);<br>void fstream::open (const char* f, int m = ios::in, int p = filebuf::openprot);<br>void fstream::open (const char* f, int m = ios::out, int p = filebuf::openprot);<br>按保护模式 p 以方式 m 打开文件 f。 | < fstream > |
| peek ()      | int istream::peek ();<br>从流中得到下一个字符或 EOF, 但不将它从流中取走。                                                                                                                                                                                                                  | < istream > |
| pow ()       | double pow (double x, double y);<br>返回 x 的 y 次方。                                                                                                                                                                                                                      | < cmath >   |
| precision () | int ios::precision ();<br>int ios::precision (int k);<br>返回流当前的精度; 第二个把当前的精度变为 k, 然后返回旧的精度。                                                                                                                                                                           | < istream > |
| tolower ()   | int tolower (int c);<br>如果 c 是一个按字典顺序的大写的字符, 返回它的小写形式, 否则返回 c。                                                                                                                                                                                                        | < cctype >  |
| toupper      | int toupper (int c);<br>如果 c 是一个按字典顺序的小写的字符, 返回它的大写形式, 否则返回 c。                                                                                                                                                                                                        | < cctype >  |

## 附录 G 十六进制数

人们通常使用基数 10 计数系统。之所以叫做十进制系统是因为希腊字 deka 的意思是“十”。我们的远古祖先通过用他们的十个手指计数而学会了它。

计算机只有两个手指（也就是说对每位仅有两个可能的值），因此二进制系统对计算机很合适。但二进制系统的问题是它们表示时需要很长位的串。例如，1996 用二进制表达为 11111001100。大多数的人都发现像这样的串处理起来困难。

如果基数是 2 的幂，二进制数很容易转化为其他的基数。例如，二进制和八进制间的转化（ $8 = 2^3$ ）只是要求把二进制的位归类为三个一组，然后把每三个一组认为是一个八进制数。例如，为了转化二进制数 11111001100，写为 11, 111, 001, 100 = 3714。这里的 11 转化为 3，111 转化为 7，001 转化为 1，100 转化为 4。从八进制转化为二进制也是很简单。例如，2650 转化为 10110101000，十进制就是 1448。注意八进制数只是使用前八个十进制数字：0, 1, 2, 3, 4, 5, 6, 7。

8 以后的 2 的下次幂是 16。使用这个基数把数字变得更短。这叫做十六进（hexadecimal）制系统（希腊语中的 hex + deka 的意思是“六” - “十”）。二进制和十六进制的转化就像二进制和八进制间的那么简单。例如，为了把二进制数 10111010100 转化为十六进制数，把这些位数归数为 4 个一组（从右到左），然后把每组翻译成相应的十六进制数：101, 1101, 0100 = 5d4。这里的 101 转化为 5，1101 转化为 11，0100 转化为 4。十六进制数 10, 11, 12, 13, 14, 15 由字母表的前六个字母：a, b, c, d, e, f 表示。

大多的操作系统提供了一个计算器应用程序，它把十六进制、十进制、八进制和二进制间的数字表示形式相互转化。例如，微软 Windows 中的计算器应用程序的位置在 Start > Programs > Accessories。在这个应用程序中，为了从十六进制转化为十进制，从它的“查看”菜单选取“科学型”，选取“十六进制”单选按钮，输入数的十六进制表示形式，然后选择“十进制”单选按钮。这里的例子（如图 G.1 所示）显示出 0x0064fdb5 的十六进制符号是 6 618 556。

输出操纵符 dec、hex 和 oct 用来转化不同的基数，就像下面的例子所示的那样。

### 例 G.1 使用输出操纵符

这里显示的是一个变量的值和地址是如何输出的：

```
int main ()
{ int n = 1492;    // 基数 10
  cout << "Base 8: n = " << oct << n << endl;
  cout << "Base 10: n = " << n << endl;
  cout << "Base 16: n = " << hex << n << endl;
```

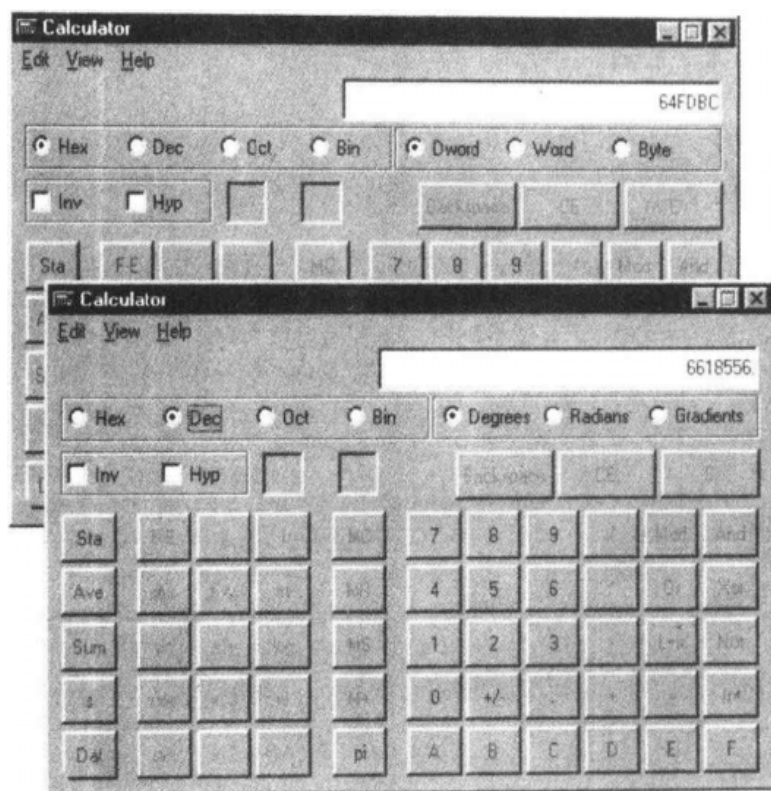


图 C.1 计算器应用程序输出

```
Base 8: n = 2724
Base 10: n = 1492
Base 16: n = 5d4
```

这里的操纵符 oct 用把下次的输出转化为八进制的形式。八进制数的表示有一个前缀 0，十六进制数的表示有一个前缀 0x。

### 例 G.2 使用输入操纵符

这里显示的是一个变量的值和地址是如何输出的：

```
int main ()
{ int r;
  cout << "Enter an octal numeral (use 0 prefix): ";
  cin >> oct >> n;
  cout << "Base 8: n = " << oct << n << endl;
  cout << "Base 10: n = " << dec << n << endl;
  cout << "Base 16: n = " << hex << n << endl;
  cout << "Enter a decimal numeral: ";
  cin >> dec >> n;
  cout << "Base 8: n = " << oct << n << endl;
  cout << "Base 10: n = " << dec << n << endl;
  cout << "Base 16: n = " << hex << n << endl;
  cout << "Enter a hexadecimal numeral (use 0x prefix): ";
  cin >> hex >> n;
  cout << "Base 8: n = " << oct << n << endl;
```

```
cout << "Base 10: n = " << dec << n << endl;
cout << "Base 16: n = " << hex << n << endl;
!
```

```
Enter an octal numeral (use 0 prefix): 0777
Base 8: n = 777
Base 10: n = 511
Base 16: n = 1ff
Enter a decimal numeral: 511
Base 8: n = 777
Base 10: n = 511
Base 16: n = 1ff
Enter a hexadecimal numeral (use 0x prefix): 0x1ff
Base 8: n = 777
Base 10: n = 511
Base 16: n = 1ff
```

### 算法 G.1 由十进制数到十六进制数

把整数  $x$  转化为它的等价的十六进制数：

1. 假设  $x > 0$ 。
2. 设置  $k = 0$ 。
3. 用 16 去除  $x$ ，设置  $x$  等于（整数）商。
4. 设置  $h_k$  等于前面除法的余数。对  $h_k$  使用 16 进制数字 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f 来表示数字 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15。
5.  $k$  加 1。
6. 如果  $x > 0$ ，重复步骤 3~6。
7. 返回  $h_k \dots h_2 h_1 h_0$ （也就是第  $j$  个十六进制符号是  $h_j$  的十六进制数）。

### 例 G.3 把十进制数 100 000 转化为十六进制数

对十进制数 100 000 应用算法 G.1，结果是  $100000_{10} = h_4 h_3 h_2 h_1 h_0 = 186a0_{16}$ ：

| k | x      | $h_k$ |
|---|--------|-------|
| 0 | 100000 |       |
| 1 | 6250   | 0     |
| 2 | 390    | a     |
| 3 | 24     | 6     |
| 4 | 1      | 8     |
| 8 | 0      | 1     |

### 算法 G.2 由十六进制数到十进制数

把十六进制整数  $h_k \dots h_2 h_1 h_0$  转化为它的等价的十进制数：

1. 设置  $x = 0$ 。



2. 设置  $j = k + 1$  (在十六进制串中的实际位数)
3.  $j$  减 1。
4.  $x$  乘以 16。
5. 另  $x$  加  $h_j$ 。
6. 如果  $j > 0$ , 重复步骤 3 ~ 6。
7. 返回  $x$ 。

#### 例 G.4 把十六进制数 f4d9 转化为十进制数

转化 f4d9 为十进制数:

| $j$ | $h_j$ | $x = 2x + h_j$                |
|-----|-------|-------------------------------|
| 4   |       | 0                             |
| 3   | f     | $16 \cdot 0 + f = 15$         |
| 2   | 4     | $16 \cdot 15 + 4 = 244$       |
| 1   | d     | $16 \cdot 244 + 13 = 3917$    |
| 0   | 9     | $16 \cdot 3917 + 9 = 62\,681$ |

因此  $f4d9_{16} = 62\,681_{10}$ 。

#### 例 G.5 把十六进制数 543ab 转化为十进制数

把 543ab 转化为十进制数:

| $j$ | $h_j$ | $x = 2x + h_j$                    |
|-----|-------|-----------------------------------|
| 5   |       | 0                                 |
| 4   | 5     | $16 \cdot 0 + f = 5$              |
| 3   | 4     | $16 \cdot 5 + 4 = 84$             |
| 2   | 3     | $16 \cdot 84 + 3 = 1347$          |
| 1   | a     | $16 \cdot 1347 + a = 21\,562$     |
| 0   | b     | $16 \cdot 21\,562 + b = 345\,003$ |

因此  $543ab_{16} = 345\,003_{10}$ 。

# 译 后 记

C++ 语言作为一种功能强大的程序设计语言，早已成为专业程序设计人员的基本设计工具之一。掌握 C++ 语言也成为国内许多高校计算机专业学生所必须的技能之一，本书正是为开设 C++ 语言课程的高校师生准备的。

本书的作者是一位有着丰富 C++ 语言经验的教师，曾经编写过多本介绍 C++ 语言的畅销书。本书由浅入深地介绍了 C++ 语言的各个方面，并在所涉及的各个知识点给出了详细的例子，使读者能够更容易了解。本书每章后都给出了大量的习题，有助于读者对所学知识进行检查。因此，无论读者是从未接触过 C++ 语言的新手，还是对 C++ 语言有一定经验的开发人员，本书都能使你对这门编程语言有全面而系统的了解。

本书主要由徐漫江、王栋、何路翻译，同时，也得到了蔡判的大力帮助，特别在此他表示感谢。

由于时间及水平所限，翻译中的错误和不妥之处，敬请读者批评指正。

译 者

# 译 后 记

C++ 语言作为一种功能强大的程序设计语言，早已成为专业程序设计人员的基本设计工具之一。掌握 C++ 语言也成为国内许多高校计算机专业学生所必须的技能之一，本书正是为开设 C++ 语言课程的高校师生准备的。

本书的作者是一位有着丰富 C++ 语言经验的教师，曾经编写过多本介绍 C++ 语言的畅销书。本书由浅入深地介绍了 C++ 语言的各个方面，并在所涉及的各个知识点给出了详细的例子，使读者能够更容易了解。本书每章后都给出了大量的习题，有助于读者对所学知识进行检查。因此，无论读者是从未接触过 C++ 语言的新手，还是对 C++ 语言有一定经验的开发人员，本书都能使你对这门编程语言有全面而系统的了解。

本书主要由徐漫江、王栋、何路翻译，同时，也得到了蔡判的大力帮助，特别在此他表示感谢。

由于时间及水平所限，翻译中的错误和不妥之处，敬请读者批评指正。

译 者

# 译 后 记

C++ 语言作为一种功能强大的程序设计语言，早已成为专业程序设计人员的基本设计工具之一。掌握 C++ 语言也成为国内许多高校计算机专业学生所必须的技能之一，本书正是为开设 C++ 语言课程的高校师生准备的。

本书的作者是一位有着丰富 C++ 语言经验的教师，曾经编写过多本介绍 C++ 语言的畅销书。本书由浅入深地介绍了 C++ 语言的各个方面，并在所涉及的各个知识点给出了详细的例子，使读者能够更容易了解。本书每章后都给出了大量的习题，有助于读者对所学知识进行检查。因此，无论读者是从未接触过 C++ 语言的新手，还是对 C++ 语言有一定经验的开发人员，本书都能使你对这门编程语言有全面而系统的了解。

本书主要由徐漫江、王栋、何路翻译，同时，也得到了蔡判的大力帮助，特别在此他表示感谢。

由于时间及水平所限，翻译中的错误和不妥之处，敬请读者批评指正。

译 者

# 译 后 记

C++ 语言作为一种功能强大的程序设计语言，早已成为专业程序设计人员的基本设计工具之一。掌握 C++ 语言也成为国内许多高校计算机专业学生所必须的技能之一，本书正是为开设 C++ 语言课程的高校师生准备的。

本书的作者是一位有着丰富 C++ 语言经验的教师，曾经编写过多本介绍 C++ 语言的畅销书。本书由浅入深地介绍了 C++ 语言的各个方面，并在所涉及的各个知识点给出了详细的例子，使读者能够更容易了解。本书每章后都给出了大量的习题，有助于读者对所学知识进行检查。因此，无论读者是从未接触过 C++ 语言的新手，还是对 C++ 语言有一定经验的开发人员，本书都能使你对这门编程语言有全面而系统的了解。

本书主要由徐漫江、王栋、何路翻译，同时，也得到了蔡判的大力帮助，特别在此他表示感谢。

由于时间及水平所限，翻译中的错误和不妥之处，敬请读者批评指正。

译 者

# 译 后 记

C++ 语言作为一种功能强大的程序设计语言，早已成为专业程序设计人员的基本设计工具之一。掌握 C++ 语言也成为国内许多高校计算机专业学生所必须的技能之一，本书正是为开设 C++ 语言课程的高校师生准备的。

本书的作者是一位有着丰富 C++ 语言经验的教师，曾经编写过多本介绍 C++ 语言的畅销书。本书由浅入深地介绍了 C++ 语言的各个方面，并在所涉及的各个知识点给出了详细的例子，使读者能够更容易了解。本书每章后都给出了大量的习题，有助于读者对所学知识进行检查。因此，无论读者是从未接触过 C++ 语言的新手，还是对 C++ 语言有一定经验的开发人员，本书都能使你对这门编程语言有全面而系统的了解。

本书主要由徐漫江、王栋、何路翻译，同时，也得到了蔡判的大力帮助，特别在此他表示感谢。

由于时间及水平所限，翻译中的错误和不妥之处，敬请读者批评指正。

译 者

# 译 后 记

C++ 语言作为一种功能强大的程序设计语言，早已成为专业程序设计人员的基本设计工具之一。掌握 C++ 语言也成为国内许多高校计算机专业学生所必须的技能之一，本书正是为开设 C++ 语言课程的高校师生准备的。

本书的作者是一位有着丰富 C++ 语言经验的教师，曾经编写过多本介绍 C++ 语言的畅销书。本书由浅入深地介绍了 C++ 语言的各个方面，并在所涉及的各个知识点给出了详细的例子，使读者能够更容易了解。本书每章后都给出了大量的习题，有助于读者对所学知识进行检查。因此，无论读者是从未接触过 C++ 语言的新手，还是对 C++ 语言有一定经验的开发人员，本书都能使你对这门编程语言有全面而系统的了解。

本书主要由徐漫江、王栋、何路翻译，同时，也得到了蔡判的大力帮助，特别在此他表示感谢。

由于时间及水平所限，翻译中的错误和不妥之处，敬请读者批评指正。

译 者

# 译 后 记

C++ 语言作为一种功能强大的程序设计语言，早已成为专业程序设计人员的基本设计工具之一。掌握 C++ 语言也成为国内许多高校计算机专业学生所必须的技能之一，本书正是为开设 C++ 语言课程的高校师生准备的。

本书的作者是一位有着丰富 C++ 语言经验的教师，曾经编写过多本介绍 C++ 语言的畅销书。本书由浅入深地介绍了 C++ 语言的各个方面，并在所涉及的各个知识点给出了详细的例子，使读者能够更容易了解。本书每章后都给出了大量的习题，有助于读者对所学知识进行检查。因此，无论读者是从未接触过 C++ 语言的新手，还是对 C++ 语言有一定经验的开发人员，本书都能使你对这门编程语言有全面而系统的了解。

本书主要由徐漫江、王栋、何路翻译，同时，也得到了蔡判的大力帮助，特别在此他表示感谢。

由于时间及水平所限，翻译中的错误和不妥之处，敬请读者批评指正。

译 者